A new approach to meeting scheduling based on mobile code

Queloz, Pierre-Antoine

UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES

Département d'informatique Professeur Christian Pellegrini

A New Approach to Meeting Scheduling Based on Mobile Code

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Pierre-Antoine QUELOZ
St-Brais (JU)

Thèse N° 3284

GENÈVE
Atelier de reproduction de la Section de Physique
2001

La Faculté des sciences, sur le préavis de Messieurs C. PELLEGRINI, professeur ordinaire et directeur de thèse (Département d'informatique), C. TSCHUDIN, professeur (Uppsala University, Department of Computer Systems - Sweden) et B. LEVRAT, professeur ordinaire (Département d'informatique), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 16 août 2001

Thèse -3284-

[illegible]

Le Doyen, Jacques WEBER

To my wonderful wife Sandra and my lovely daughter Soline.

Contents

Preface

First of all, I would like to thank my advisor, Prof. Christian Pellegrini, who guided this research, during the five years I could spend in his group. His support, his precious advice and his encouragement made this period of my live an extremely fruitful and interesting one. Next I would like to thank Dr. Lamia Friha, who initiated the ISACOM project and accompanied me during the first half of this research. She guided my discovery of the scheduling domain and of the world of multi-agent systems, proposed interesting problems to think about and I had much pleasure working with her. I have also learned a lot from Prof. Christian Tschudin, who defined the ISACOM project as well. His approach of mobile code has been a solid starting point for all my experiments. I also thank him to participate in my thesis jury. I am grateful to Prof. Bernard Levrat, who influenced the course of my professional path several times. Being his assistant for the sofware engineering course was a very good experience and had a significant positive impact on the developpment of the ideas presented in this thesis. His interest for this work and his participation in the jury are a pleasure and a reward for me. Thank you Alex Villazón, for the inspiring work we did together, the good time we had and the possibility to share all kinds of concerns. Thank you also to Dr. Danuta Sosnowska, and Dr. Jan Vitek. Although we were working on different projects, we had a couple of occasions to compare them and it allowed me to expand the horizon of my own work. Special thanks to Dr. Jarle Hulaas and Dr. Ciarán Bryce, for their comments on the early versions of this dissertation. They helped me a lot to pinpoint the major ideas and hopefully to express them more clearly. Last I would like to thank all the colleagues of "Café 332" who became excellent friends and made all this time spent at the University exciting and enjoyable.

Note: The pronoun *he* and the possessive form *his* are used in places throughout the book. They are used in the general sense of anyone and connote no gender.

Résumé

La thèse et son contexte

Le but principal de ce mémoire est de montrer comment de nouvelles applications informatiques, facilement accessibles par l'intermédiaire de la toile (*World Wide Web*), peuvent être construites sur la base d'un environnement d'exécution pour code mobile. La possibilité de déplacer des programmes, et de choisir lors de l'exécution sur quel ordinateur d'un réseau une certaine opération va être effectuée, offre de nouvelles possibilités aux concepteurs de logiciels. Nous voulons comprendre dans quel contexte pratique cette flexibilité supplémentaire peut être exploitée, comment elle influence notre manière de concevoir l'architecture de ces logiciels, et quels avantages peuvent être attendus de la mise en oeuvre de cette nouvelle technique.

Nous définissons un code mobile comme un programme qui est transféré d'un ordinateur à un autre, qui est exécuté à sa destination, d'où il est capable d'effectuer d'autres transferts et de lancer de nouvelle exécutions. Cette définition englobe les concepts plus spécifiques de **messagers** (un fragment de code envoyé avec un paquet de données sur une plate-forme d'exécution distante), d'**objet mobile** (une instance d'une classe qui se déplace avec son état, et si nécessaire son code) et d'**agent mobile** (un objet mobile avec son propre *thread* d'exécution).

Selons nous, la caractéristique la plus intéressante de la technique du code mobile est qu'elle permet l'encapsulation de protocoles. En effet, l'échange de code mobile permet de réaliser les mêmes protocoles de communication que l'échange de messages, avec l'avantage de pouvoir transmettre le code implémentant de nouveaux protocoles en temps voulu. Nous pensons que cet avantage peut être extrêmement profitable, lorsqu'il permet d'éviter la définition de standards et de minimiser le nombre de conventions pré-établies dans la communication entre logiciels. Il est en effet très difficile de définir de bons standards pour la communication et l'inter-opération de logiciels. Implémenter ces standards pose également de gros problèmes, et les résultats en

terme de compatibilité laissent souvent à désirer. Standards et protocoles sont également quasi impossibles à changer, une fois qu'ils sont adoptés à plusieurs endroits et toute modification risque de compromettre la compatibilité de ce qui fonctionne déjà. Ces difficultés ne peuvent être ignorées dans le contexte actuel de l'informatique et des télécommunications, où l'innovation est permanente et les progrès fulgurants.

Nous montrons comment utiliser le code mobile de manière judicieuse, de sorte que des questions d'implémentation importantes puissent être laissées ouvertes, et les fonctionnalités correspondantes fournies par des programme extérieurs, développés par d'autres personnes. Ainsi, si un logiciel est conçu dès le départ pour être étendu, il n'y a pas besoin de prévoir tous les cas d'utilisation, mais seulement une petite fraction qui représente les situations les plus courantes. On évite ainsi d'encombrer le programme avec des fonctionnalités inutilisées, et on réduit les problèmes de collaboration qui peuvent se poser lors de la conception. Au lieu de construire des logiciels qui ne résoudent pas le bon problème à cause du manque de renseignements, on laisse venir de l'extérieur (là où les besoins sont mieux compris) les programmes adaptés aux contextes et utilisations particuliers qui peuvent se présenter.

Systèmes d'information distribués

Parmi les quelques domaines d'applications dans lesquels le code mobile est considéré comme potentiellement intéressant (p. ex. recherche d'informations, gestion de réseaux, calcul parallèle, Applets) ce travail se concentre sur celui des systèmes d'information distribués. Nous commençons par décrire pourquoi ces systèmes sont généralement difficiles à concevoir et à faire fonctionner de manière fiable.

Tous les systèmes informatiques sont menacés par les changements très rapides qui se produisent autour d'eux. Ce sont tout d'abord les changements du monde lui-même : les personnes se déplacent, des objets sont créés, échangés, détruits, des événements se produisent, les modèles scientifiques sont raffinés, etc. Pour être utiles, la plupart des systèmes doivent prendre en compte cette **dynamicité** et la refléter dans l'information qu'ils contiennent et qu'ils offrent. Nous pouvons souvent anticiper ces changements en effectuant une bonne modélisation de la partie du monde sur laquelle notre système est basé. Il faut encore que lorsqu'un événement se produit le système en soit informé et qu'il soit mis à jour, ce qui demande souvent une intervention humaine mais devrait tant que possible être fait automatiquement, soit pour des raisons de coût, soit simplement pour éviter des erreurs.

Il est beaucoup plus difficile d'anticiper de quelle manière l'utilisation d'un système va évoluer, mais il est notoire que les utilisateurs inventent toujours de nouvelles manières d'exploiter l'information et que l'environnement technique est en perpétuelle mutation. Cela a pour effet de pousser les systèmes au delà de leurs limites. L'expérience montre qu'un système qui n'est pas conçu pour **évoluer** et prendre en compte ces besoins imprévisibles au moment de sa conception finit par être inutilisable. À l'intérieur d'une entreprise cela représentera de gros frais pour développer un nouveau système. Dans un contexte commercial, cela peut résulter dans l'abandon du système au profit de concurrents mieux adaptés.

Tout se complique encore quand un système est **distribué**. Il faut alors faire face à des problèmes nombreux et variés. Mentionnons par exemple les problèmes de compatibilité, au niveau du matériel, des systèmes d'exploitation ou des langages de programmation. De nouveaux problèmes apparaissent également pour effectuer des mises à jour de logicel. Le système peut aussi être victime de pannes partielles, de coupures de réseau, de temps de latence imprévisibles, etc. On peut aussi se trouver confronté à des problèmes de collaboration entre ceux qui offrent un service et ceux qui l'utilisent soit parce que le système s'étend au-delà des limites d'une organisation (buts différents), soit parce que les utilisateurs sont trop loin des développeurs (dans l'espace ou pire : dans le temps), soit encore parce que les utilisateurs sont nombreux et ont des besoins contradictoires.

Nous pensons donc que ces trois caractéristiques: dynamicité, évolution et distribution sont la cause de nombreux problèmes. Nous utiliserons l'acronyme DEDIS (*Dynamic Evolving Distributed Information Systems*) pour désigner les systèmes d'information présentant ces caractéristiques, et pour lesquels l'utilisation de code mobile semble particulièrement judicieuse.

Aspects techniques du logiciel

D'un point de vue plus théorique nous pouvons expliquer pourquoi les DEDIS sont si difficiles à construire par le grand nombre d'aspects techniques différents qu'il faut prendre en compte dans leur développement. Non seulement chacun de ces aspects requiert un traitement particulier dans le programme, mais il existe de nombreuses dépendances entre aspects. Ces dépendances nous empêchent de traiter les aspects indépendamment les uns des autres et sont à la source du phénomène de *cross-cutting*. Ce terme désigne le fait qu'il est quasi impossible de décomposer le programme en modules qui traitent les aspects séparément, et d'éviter que le code qui traite un aspect ne se répande dans les modules écrits pour traiter les autres aspects.

Bien que la difficulté de traiter un grand nombre d'aspects soit connue depuis longtemps, et que des techniques pour éviter le problème de *cross-cutting* soient en cours de développement, nous n'avons pas pu trouver d'inventaire complet des aspects. Nous avons donc effectué un travail de catalogage et de description des aspects et de leurs relations qui est résumé dans la Fig. 0.1, p. 10.

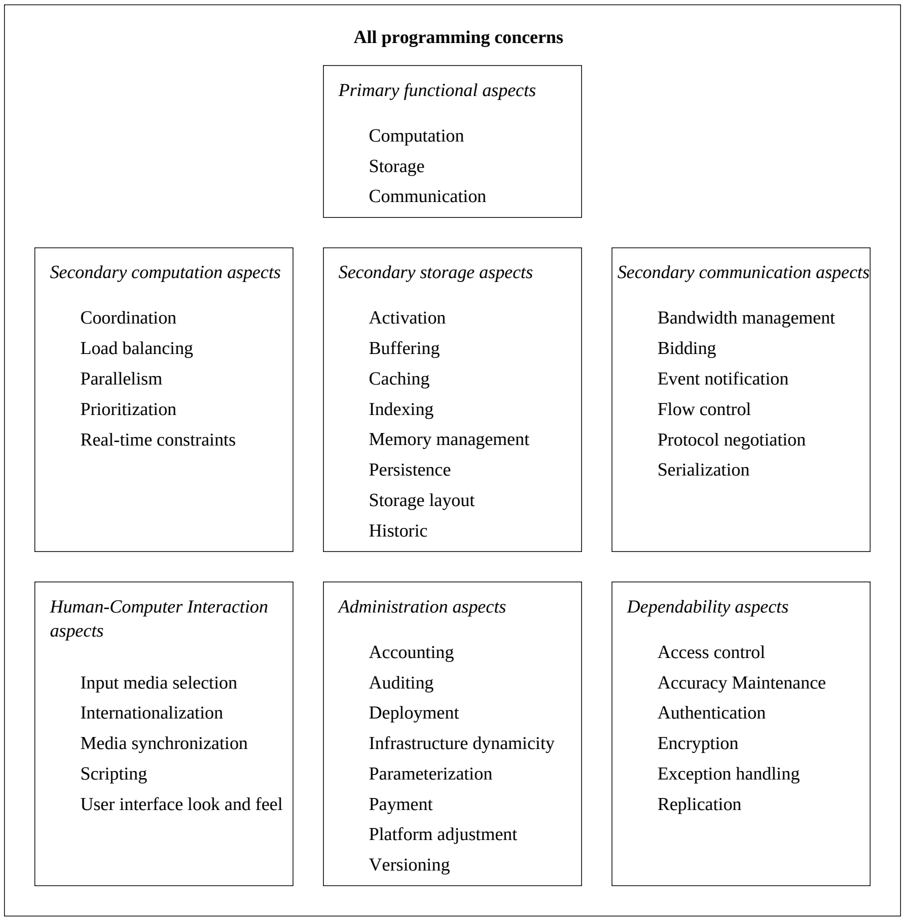


Figure 0.1: Une tentative de classification des aspects. Noter cependant que plusieurs aspects pourraient trouver leur place dans plus d'une catégorie.

Notre catalogue recense 41 aspects au total, dont trois aspects particuliers dits “fonctionnels” et 38 aspects non-fonctionnels. Les trois aspects fonctionnels sont les trois opérations fondamentales de l'architecture de von Neumann, et de la machine de Turing, son équivalent théorique: calcul, mémorisation, communication. Cette définition correspond également à l'idée de programmation fonctionnelle, dans laquelle les autres aspects sont considérés comme des questions secondaires d'implémentation, et généralement ignorés.

Lorsqu'un programme s'exécute, il doit prendre en compte tous les aspects fonctionnels et non-fonctionnels, il faut donc préalablement implémenter les spécifications non-fonctionnelles à l'aide des trois opérations fonctionnelles de base. Ce travail de traduction est généralement fait par un programmeur, mais peut également parfois être fait par un compilateur (*aspect weaving*).

Notre principale hypothèse est que dans un environnement d'exécution de code mobile, une majorité des aspects sont plus faciles à traiter, et n'interfèrent pas de manière aussi délicate, parce qu'une nouvelle répartion des responsabilités est possible. Habituellement, celui qui offre un service doit implémenter la totalité des fonctionnalités requises par différents types d'utilisateurs et d'applications, ainsi que par divers environnements matériels. Avec le code mobile, il devient possible de déléguer certaines responsabilités à ceux qui savent exactement quelle fonctionnalité est nécessaire. Par exemple aux développeurs d'une application avec laquelle le système doit interagir, ou avec des “développeurs d'extensions” mandatés par les utilisateurs. Les applications qui en résultent sont potentiellement moins coûteuses à développer et plus faciles à maintenir. De plus, elles sont moins sujettes aux conflits habituels entre généralité et bonnes performances, qui sont deux qualités contradictoires, donc difficiles à obtenir en même temps.

Afin de vérifier cette hypothèse, nous avons d'une part réexaminé les avantages du code mobile déjà décrits dans la littérature à la lumière de notre nouvel inventaire des aspects. D'autre part, nous avons implémenté un nouveau service, entièrement utilisable, en exploitant l'un des nombreux environnements pour code mobile existants. On recense actuellement 70 telles plate-formes d'exécution (plus ou moins utilisables), dans lesquelles la la possibilité de déplacer des programmes est prévue, en plus des moyens standards de communication par échanges de messages. La plupart de ces plate-formes sont basées sur le langage Java et sur le paradigme de programmation orientée-objets. Pour notre travail, nous avons suivi cette tendance et la majorité des résultats présentés adoptent cette technologie. Cependant l'idée de code mobile peut également être exploitée indépendamment de l'orientation-objets.

Cette étude a révélé des simplifications diverses pour deux tiers des aspects, lorsque le code mobile est utilisé pour assembler des parties immobiles d'un système distribué, comme schématisé par la Fig. 0.2, p. 12.

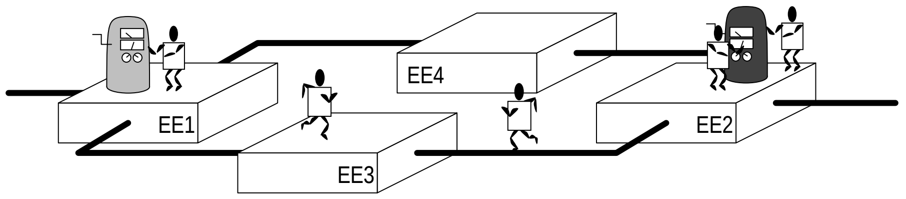


Figure 0.2: Deux services (les “machines” grises arrondies munies de boutons et de cadrans) qui interagissent au moyen de code mobile, et qui sont installées dans un réseau de quatre plate-formes interconnectées (boîtes blanches).

Les simplifications que nous avons observées peuvent être regroupées en trois grandes catégories: (1) les aspects qui peuvent être délégués aux clients selon la nouvelle répartition de responsabilités que nous avons déjà présentée ; (2) les aspects pour lesquels le client bénéficie de la proximité, parce que les composants mobiles peuvent interagir directement avec le service, sans que le réseau n'interfère ; (3) les aspects qui sont mieux supportés par certaines plate-formes d'exécution de code mobile existantes que par les systèmes d'exploitation habituels.

La somme de ces petits bénéfices représente une très forte motivation d'exploiter le code mobile dans le contexte des DEDIS, même si nous ne pouvons pas trouver une application emblématique, qui ne serait possible qu'avec le code mobile. À la lumière de nos résultats, cela semble même improbable qu'une telle application existe, puisque les bénéfices du code mobile sont rattachés aux aspects non-fonctionnels. Il n'y a donc pas à proprement parler de nouvelle fonctionnalité qui ne pourrait pas être implémentée avec des modèles de programmation classiques et des communications par échange de messages, seulement des environnements et des mécanismes qui facilitent la conception du système.

<u>Sélection de dates</u>

Pour notre étude pratique, nous avons choisi de travailler sur le problème de la sélection de dates. Nous pouvons énoncer ce problème de la manière suivante :

> Permettre à un groupe de personnes de collaborer pour déterminer un intervalle de temps pendant lequel une activité prévue doit avoir lieu.

Notons que les personnes impliquées dans la décision ne participent pas forcément à l'activité, il ne s'agit donc pas seulement de trouver un moment où les personnes n'ont pas d'autre rendez-vous. D'autre part, notre énoncé ne fait aucune supposition quant aux raisons qui peuvent les inciter à choisir un intervalle plutôt qu'un autre. Elles préférereont peut-être certains moments de la journée, ou bien cela dépendra de ce qu'elles ont prévu de faire avant ou après, de ce qu'elles doivent préparer, etc.

Nous avons envisagé ce problème comme un problème de décision, dans lequel l'informatique facilite la communication et la représentation des contraintes, mais où le choix final sera toujours laissé à l'utilisateur. Il n'est pas question ici d'automatiser complètement la résolution du problème car il est trop complexe, mais de proposer un service, qui assiste l'utilisateur et lui permet de gagner du temps.

D'autres chercheurs ont essayé d'automatiser la résolution par des mécanismes de négociation. Il faut alors beaucoup d'ingéniosité pour représenter les multiples facteurs (entre autres sociaux) qui peuvent entrer en jeu dans une telle opération et arriver à un comportement qui satisfait l'utilisateur. Ce dernier doit de plus faire de gros efforts pour configurer suffisamment le logiciel.

Caractéristiques de ce problème

Les trois caractéristiques des DEDIS (dynamicité, évolution, distribution) sont présentes dans le problème que nous voulons résoudre.

La facette de **distribution** est la plus évidente : chaque personne a des préoccupations différentes et elle gère son emploi du temps de la manière qu'elle préfère. Elle utilise peut-être un ordinateur ou un agenda électronique pour gérer son calendrier, mais pas forcément. Un système résolvant ce problème doit donc avant tout aider les personnes à communiquer et être capable de prendre en compte l'hétérogénéité des moyens utilisés.

La **dynamicité** est également très présente : les activités de chacun se modifient constamment, tout en influençant les possibilités futures. Ainsi nous n'aurons jamais une solution définitive, mais il faudra en permanence tenir compte de l'évolution de la situation et vérifier que les nouvelles contraintes ou décisions n'entrent pas en conflit avec des choix antérieurs. Deux difficulté peuvent survenir : collision entre deux activités, due au temps qu'il faut pour trouver une solution ; impossibilité de trouver une date, qui exige de décommander une activité. Nous avons réfléchi à ces problèmes et expliquons comment nous pouvons les traiter.

Finalement l'**évolution** se manifeste par la nécessité de faire fonctionner le service avec les futures technologies de communication. Nous voulons également être capables de l'associer à de nouvelles applications, par exemple à un système faisant des réservations de salles, ou aux multiples systèmes de calendrier personnels disponibles actuellement. Enfin nous permettons aux utilisateurs d'automatiser certaines tâches, par exemple placer des réponses automatiques lorsqu'ils sont en vacances.

Notre solution

Notre solution suppose que l'organisateur et les participants disposent tous d'un accès à la toile et à la messagerie électronique de l'internet. Nous supposons également qu'ils lisent régulièrement leurs messages. Cela a une certaine importance car c'est par ce moyen qu'ils communiquent. L'organisation d'une réunion peut être résumée en 4 étapes principales :

1. l'organisateur de la réunion se connecte à notre site, il crée une nouvelle réunion, donne une description et une liste d'adresses (celles des participants) ; il propose un ensemble de dates et d'heures qui lui semblent possibles pour cette réunion ; il demande au système d'envoyer des invitations aux participants ;

2. lorsqu'ils reçoivent le message d'invitation, les participants se rendent également sur le site, où une page spécifique à la réunion (et à chaque participant) permet de répondre à l'invitation ; pour chacune des dates et heures proposées par l'organisateur, le participant indique si elle lui convient ou pas, en fonction de ses préférences et de son propre emploi du temps ;

3. l'organisateur revient sur notre site pour choisir une date ; le système l'aide à visualiser les contraintes introduites par les participants et à prendre sa décision ;

4. les participants reçoivent un message de confirmation lorsque la date est déterminée.

Nous proposons donc une solution informatique à ce problème, qui est basée sur la toile et le courrier électronique. Par rapport aux moyens traditionnels (téléphone, fax, courrier) les avantages pour les personnes organisant la réunion sont multiples : les moyens de communication sont rapides et bon marché, l'expression des contraintes est facilitée par l'utilisation de formulaires, la recherche de la solution est facilitée par une visualisation claire des possibilités, la recherche de conflits est facilitée par une recherche rapide des intersections.

Par rapport aux autres logiciels existants pour résoudre ce problème de choix de dates, notre approche a les spécificités suivantes : elle permet à chaque utilisateur d'utiliser n'importe quel type d'agenda, alors que certaines solutions lui imposent de répertorier toutes ses activités sous format électronique ; elle garantit la confidentialité puisque les raisons d'accepter ou de refuser une date ne sont pas communiquées ; elle facilite la communication et aide l'organisateur à choisir mais ne prend aucune décision de manière autonome ; elle donne accès à l'information depuis n'importe quel ordinateur relié à l'internet ; elle est extensible par du code mobile afin de faciliter l'intégration future avec d'autres systèmes et le maintien de la cohérence de l'information à l'intérieur du service, ainsi que des informations externes qui en dépendent.

Architecture

Dans la Fig. 0.2, p. 12 nous avons utilisé le terme "service" pour désigner les composantes immobiles. Nous voulons par là marquer la différence par rapport aux applications habituelles, qui sont moins accessibles et plus difficiles à intégrer dans de nouveaux systèmes. La majorité des programmes sont difficiles à déplacer, parce qu'ils ne peuvent bien fonctionner que dans un contexte particulier, ou sont rattachés à d'autres programmes ou à de grandes quantités de données–penser à un système qui sert une fonction bien précise dans une grande entreprise. On ne peut donc pas simplement les prendre et les installer partout où leurs fonctonnalités sont nécessaires (par exemple dans un autre grand système qui est lui-aussi difficile à déplacer). Mais dès que ces programmes sont facilement accessibles, par exemple avec du code mobile, ils deviennent des composants réutilisables pour des systèmes à grande échelle.

Dans notre approche, les informations et les fonctionnalités des services sont accédées par des composants mobiles et de petite taille, qui sont autorisés à s'exécuter sur les mêmes machines que les services. Pour le transfert d'informations et d'événements entre services et extensions, nous n'avons pas eu besoin de protocoles ou d'ontologies plus complexes que le modèle de programmation orientée-objets: l'information est encapsulée dans des objets, qui peuvent être référencés, et dont les méthodes, qui ont des noms bien définis, peuvent: (1) être appelées, avec des paramètres, (2) retourner des résultats, (3) lancer des exceptions.

Une partie conséquente du mémoire décrit l'architecture de tels services, ainsi que les mécanismes que l'environnement d'exécution doit offrir, afin que l'approche soit possible en pratique. Cette architecture se base sur les techniques récentes de conception orientée-objets, qui doivent toutefois être quelque peu adaptées à la nouvelle situation. La Fig. 0.3, p. 16 schématise cette architecture, qui est également celle que nous avons choisie pour implémenter le service de choix de dates.

- La base de données peut être très grande. Il est donc exclu de l'avoir entièrement en mémoire vive. Les classes du *package* "Managers" encapsulent l'accès à la base.

- Les classes du *package* "Service interface" contiennent la logique du service. Ce sont elles qui donnent un sens à la base de données et qui offrent une interface pour les composants mobiles. La difficulté consiste à offrir

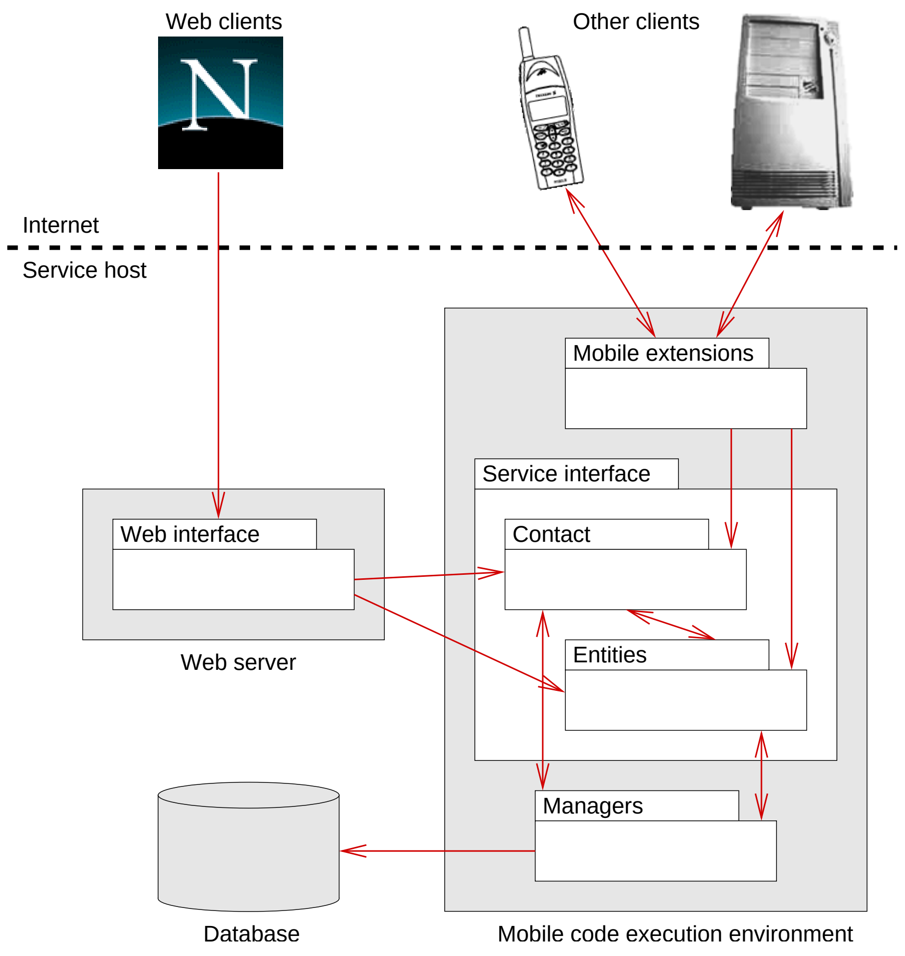


Figure 0.3: Architecture générale pour un service fonctionnant à l'intérieur d'une plate-forme pour code mobile. Les "dossiers" représentent des *packages* de composants. Les flèches représentent les dépendances et interactions possibles entre ces composants.

un accès qui soit facile à comprendre, et qui ne cache pas les informations importantes et les événements intéressants du service.

- Du moment que le service est implémenté dans un environnement de code mobile, des composants de provenances diverses peuvent venir s'exécuter sur la même machine. Les mécanismes nécessaires sont peu nombreux : un contrôle des droits d'accès qui préserve la confidentialité, un contrôle de l'utilisation de ressources qui empêche la sur-consommation, des mécanismes de notfication pour les événements internes au service, et pour informer les composants "étrangers" lorsque le service va cesser de fonctionner. Nous montrons que cela représente peu d'efforts supplémentaires pour le créateur du service et suffit pour le rendre tout à fait extensible.

- Un autre grand avantage de la toile que nous tenons absolument à exploiter est la quasi universalité de l'interface utilisateur. À l'heure actuelle, un document HTML peut être affiché par tout ordinateur relié à l'internet, sans qu'il soit nécessaire d'y installer de nouveau logiciel. Certaines optimisations sont possibles en utilisant des Applets ou des scripts qui s'exécutent sur le "client" mais nous éviterons de le faire car cela pose souvent des problèmes de compatibilité. L'interaction avec l'utilisateur sera donc gérée par les classes du *package* "Web interface".

Résumé des contributions

Les contributions originales de cette thèse sont les suivantes:

1. Description des aspects techniques du logiciel et construction d'un premier catalogue, aussi complet que possible. A notre connaissance, un tel travail n'avait jamais été réalisé et de nombreuses notions relatives aux aspects étaient utilisées sans véritables définitions.

2. Description de l'encapsulation de protocoles par le code mobile, et illustration par de nombreux exemples. Cette notion n'est pas nouvelle, mais peu d'autres travaux lui ont accordé l'attention qu'elle mérite.

3. Description d'un domaine d'application dans lequel le code mobile est particulièrement intéressant (DEDIS), des problèmes que cela permet de résoudre (intégration, qualité de l'information) et justification par le grand nombre d'aspects pris en compte par cette technique. L'idée d'utiliser le code mobile dans le domaine des systèmes distribués n'est pas nouvelle, mais l'approche "par les aspects" donne de nouveaux critères pour évaluer l'impact du code mobile.

4. Architecture pour des services ouverts et extensibles. La plupart des travaux sur le code mobile s'intéressaient à ce qui bouge, nous proposons un modèle plus détaillé de la partie immobile et étudions les mécanismes qui lui permettent d'interagir avec des extensions mobiles, et de gérer au mieux le grand nombre d'aspects qu'on trouve dans les DEDIS.

5. Pour notre étude de cas nous avons implémenté un nouveau service pour choisir une date de manière collaborative. C'est un outil utile, mis à disposition gratuitement pour tous les utilisateurs de l'internet.

6. Pour implémenter notre service en Java, nous avons dû développer une nouvelle librairie de classes pour la gestion de calendriers, d'intervalles et de domaines temporels. Cette librairie est décrite en détail et pourra être réutilisée.

7. Présentation détaillée de plusieurs extensions concrètes de notre service (agents mobiles dans l'environnement "Voyager"). Qui illustrent bien l'intérêt de l'encapsulation de protocoles, et montrent quelles conventions sont nécessaires entre le fournisseur du service, et ceux qui programment les extensions, toujours dans la perspective de gérer aux mieux les différents aspects techniques.

Abstract

The main goal of this dissertation is to show how useful real-world applications, easily accessible from the Web, can be built on top of mobile code platforms. Being able to move programs, and to choose at run time on which network host a given computation will take place, provides a new degree of freedom to software designers. We want to understand in which practical context this additional flexibility can be exploited, what could be a good design based on this new technology, and what benefits should be expected from its adoption.

Our work emphasizes the idea that the essential interest of mobile code is the ability to encapsulate protocols, and that one of the most profitable usage is to reduce the need for standards and pre-established conventions. Standards for networking or application inter-operability are notoriously hard to concieve, their implementation often requires a lot of work, and the resulting inter-operation is seldom satisfactory. Moreover, such protocols are extremely difficult to change without compromising compatibility once they are deployed, a significant problem, given the amazing rate of innovation and unpredictability of the user needs in the field of computer technology. With appropriate use of mobile code, we can delay some important decisions, and implement in our software support for only a small fraction of the potentially infinite number of situations in which it will be used, and let others provide additional functionality, that perfectly matches their specific requirements. Not only, we avoid to clutter our programs with unnecessary features, but we also reduce the needs for collaboration, and avoid to end up with unsatisfactory solutions because of lack of information.

Among the few application domains in which mobile code technology could be harnessed (e.g. information retrieval, network management, parallel processing, Applets) this work will concentrate on large-scale distributed information systems. We first study the characteristics of these systems that make them difficult to develop and to operate: (1) the *dynamicity* of the environment and of the computing infrastructure compromise integrity and availability of information; (2) constant *evolution* of the systems, flexibility and easy integration are all required to follow the frequent changes in the user's needs and technical context; (3) administrative *distribution* makes collaboration difficult and fosters conflicting objectives; (4) physical distribution has inherent technical traps. We use the acronym DEDIS to define these Dynamic, Evolving, Distributed Information Systems, for which mobile code seems to be perticularly well suited.

From a more theoretical perspective, we explain these difficulties by the large number of different technical aspects that must be taken into account in this kind of software. Having established an extensive inventory of these aspects, and of their dense graph of dependencies, we are able to explain why developing large-scale systems is so challenging: because there is a far greater number of aspects that must be taken into account and "woven" together in the resulting software than in desktop applications.

Our main hypothesis is that within a mobile code environment, a large fraction of aspects are easier to take into account, and don't interfere in the same way, because a new repartition of the responsibilities is possible. Usually, a service provider must implement all the functionality required by different kinds of users, external applications, and physical environments. With mobile code, it becomes possible to shift responsibilities towards those who know what functionality is really needed, for instance developpers of related applications, or extension programmers working on behalf of end users. The resulting applications are potentially less expensive to develop, easier to maintain, and additionally, they don't suffer from the common problem that generality and good performances are difficult to achieve at the same time.

In order to check this hypothesis, we have on one hand categorized benefits described in the mobile code literature according to our new inventory of aspects; on the other hand, we have implemented a new fully-functional service within one of the numerous existing mobile code execution environment. This study has revealed various levels of simplifications for two thirds of the aspects, when mobile code is used to "glue" together immobile parts of a distributed system. The simplifications we have observed can be grouped in three broad categories: (1) aspects that can be delegated to the client, thanks to the shift of responsibility that we have already discussed; (2) aspects where the clients benefits from locality, because mobile components can interact with the service without interference from the network inbetween; (3) aspects that are better supported by existing mobile code execution environments than by usual operating systems.

The sum of these little benefits represents a very strong incentive to use mobile code in the context of DEDIS, even if we cannot exhibit a "killer application" that would be possible only with mobile code. In the light of our results, it even seems improbable that such an application exists, since the benefits of mobile code are related to the non-functional aspects (often considered as secondary “quality of service” issues), and not directly enabling a new kind of function that cannot be implemented with classical programming models and communication by message passing.

For our case study, we have chosen to work on the meeting scheduling problem, which is a good candidate for the study of DEDIS. Our system lets a group of people collaborate in order to determine at what time a forthcoming activity will take place. This requires communications, computations, taking into account a variety of calendaring hardware and software, being able to dynamically take into account new constraints, and being able to adapt to new technical requirements, and new habits of the end users.

A detailed study of the state of the art in scheduling and constraint satisfaction reveals that our solution has several advantages in comparison with other approaches for date selection. It makes this complex decision processes very easy but, and at the same time doesn’t require users to change their calendaring habits, nor to behave in a very disciplined way. Moreover, there is no risk to be stuck with a product that doesn’t evolve in the right direction since it offers only a well delimited functionality and can easily be integrated with future technologies or applications.

We name our software a “service” in order to emphasize the difference with conventional applications that are less accessible and harder to integrate in a larger system. It is implemented as an immobile component within a mobile code environment environment. Its informations and functionalities can be accessed by mobile components that are allowed to run on the same computer. The transfer of informations and events between service and extensions doesn’t require more complex protocols or ontologies than the object-oriented programming model: data is encapsulated within objects, that can be referenced and whose methods, which have well defined names, can be called with parameters, return values and throw exceptions.

An important part of the dissertation describes this new approach, the mechanisms that the execution environment must provide in order to make it practically feasible, and how the service itself must be structured. The architecture of the service is based on recent object-oriented techniques, which must somewhat be adapted to the new situation of having foreign processes, with their own state and programs, running on the same host as the service, and needing to interact with it in order to satisfy the needs of many unknown users, without requiring collaboration with the service provider.

Part I
Composition of services with mobile code

In this first part of the thesis we describe some shortcomings in today's technology for distributed information systems and we propose an approach based on mobile code that has the potential to overcome some of these shortcomings. We describe an architecture for open services that will be illustrated by the case study of the second part.

Chapter 1

The thesis and its context

Chapter highlights

- The essential topic of this research: building software applications on top of mobile code platforms.
- The new perspective that justifies the study: no “killer application” has been found for mobile code because the biggest potential of this approach lies in the treatment of non-functional requirements, which are usually considered as secondary, or ignored, in small scale, proof-of-concept experiments.
- The potential impact of the research: improve the quality of large-scale software systems, thanks to a better understanding and new techniques, especially in the treatment of non-functional aspects.
- The characteristics that make large-scale systems difficult to build: the dynamicity of the environment and of the computing infrastructure compromise integrity and availability of information; constant evolution of the systems, flexibility and easy integration are required to follow the frequent changes in the user’s needs and technical context; administrative distribution makes collaboration difficult and fosters conflicting objectives; physical distribution cannot be ignored because of its inherent technical traps.
- The promise of the messenger paradigm: communication by exchange of programs instead of messages offers the huge benefit of reducing the number of necessary conventions, and allows the encapsulation of non-

functional aspects, in order to decouple components from the context in which they are used.

- The need to learn building large-scale systems with mobile code: it is not a recent idea but it has not yet been studied thoroughly because researchers concentrated on the implementation of platforms and on solving the intrinsic pitfalls of the new technology.

1.1 Main statement and situation

Useful real-world applications, easily accessible from the Web, can be built on top of mobile code platforms and the flexibility thus obtained allows the satisfaction of several important needs that are difficult to handle otherwise.

Mobile code and related technologies have been the subject of much research and industrial interest for at least one decade but only a handful of real-world applications have been described, and widespread use has yet to occur. The goal of this dissertation is to show that there are indeed many important applications that can greatly benefit from the ability of mobile code to choose, at run time, on which network host a given computation must take place.

We explain the lack of "killer application" for mobile code by the fact that the interesting aspects where mobile code has the biggest impact are not the primary functional aspects of software design, but are rather seen as quality of service aspects. This means that alternative solutions often seem more straightforward. And that for almost all software developments, in which these aspects are either played down or absent, the need to use mobile code has not been felt as strongly as it could be.

However, looking at the current state of the software world reveals big imperfections like lack of robustness, high maintenance costs, etc. Our hypothesis is that these imperfections and problems may result from insufficient attention given to some of the non-functional aspects that mobile code could precisely handle.

Throughout the dissertation, we will mainly use the general concept of **mobile code**, which encompasses a few more specific notions like:

- **messenger**—a code fragment shipped with a piece of data to a remote execution platform (§ 3.5.1, p. 69),
- **mobile object**—an instance of a class that moves with its state, and if required its code (§ 3.3, p. 56),
- **mobile agent**—a mobile object with its own thread of execution (p. 83).

In this chapter we will define the context of our research, and try to identify the characteristics of these applications that could potentially benefit from new architectures based on mobile code.

1.2 Large-scale systems and DEDIS

In (Committee on Information Technology Research in a Competitive World, 2000), a recent report published by the US National Academy Press (NAP) that tries to define important directions for the research in computer science and information technology (IT), a large group of experts has identified the inability to build large-scale systems as one of the most urgent challenges. Their definition of a large-scale system is the following:

> *Large-scale systems are IT systems that contain many (thousands, millions, billions, or trillions or more) interacting hardware and software components. They tend to be heterogeneous—in that they are composed of many different types of components—and highly complex because the interactions among the components are numerous, varied, and complicated. They also tend to span multiple organizations (or elements of organizations) and have changing configurations. Over time, the largest IT systems have become ever larger and more complex, and, at any given point in time, systems of a certain scale and complexity are not feasible or economical to design with existing methodologies.*
>
> (Committee on Information Technology Research in a Competitive World, 2000)

The keywords of this definition, which can be viewed as the characteristics that make this kind of systems extremely hard to design and operate are **scale**, **physical distribution**, **heterogeneity**, **administrative distribution** and **dynamicity**. In (Queloz and Pellegrini, 1999) we stated that a favourable context for the application of mobile code is **dynamic**, **evolving**, **distributed** information systems, and we proposed the acronym DEDIS to name such systems. In the light of the NAP report we realize that our objectives were very ambitious, but aimed in the right direction.

1.3 Dynamicity

In order to build a system that will work for a long period of time we must be able to **anticipate** what is going to happen around it. The first kind of dynamism we have to deal with is the dynamicity of the world itself: people are moving, objects are created, exchanged, destroyed, new books are published, scientific research refines its models, etc. To be useful most systems must take these changes into account and reflect them in the information they contain and offer. We can anticipate some of these changes if we build a good model of the world on which we base our system. Yet, when an event occurs, the system must be informed and its state must be updated. Too often human intervention is required, although it could be better to update relevant documents or databases automatically, either for cost reasons or to prevent oversight.

Broadly speaking most systems that handle events and world dynamicity cope by applying the principle of **publish and subscribe**. This principle of publish and subscribe for event notification is so general that it was already in use long before computers existed. It is the natural way to pass information to interested parties and there is no wonder that it is present in so many different contexts in today's information systems, from electronic mailing lists to "push channels" and inside object-oriented applications underlying the "Observable-Observer" Design Pattern.

We call **accuracy maintenance** this first aspect of dynamicity. Failing to handle this technical aspect properly leads to invalid information and a system that provides misleading results which may incur cost and frustration for the users.

The second related aspect is **infrastructure dynamicity**: the fact that components and devices are added to or removed from the system's environment. In a large-scale system, sub-systems cannot expect to have permanently established connections with all others. It is a consequence of failures, upgrades and transformations. Furthermore, end users are not permanently running all their applications and leaving all their devices switched on. Failing to address the resulting disconnections or the appeareance of new hardware and software components may lead to loss of functionality, poor resource usage, or to failures when changes are not taken into account and the environment responds in an unexpected way.

1.4 Evolution

It is extremely difficult to anticipate how people are going to use a successful information system. If the system has enough users, it is almost certain that they will invent new ways of using it and discover new requirements that were completely unpredictable. To satisfy these new needs, the system must be able to evolve. Experience shows that systems that cannot evolve and take into account these unforseeable needs quickly become obsolete. Inside a company, developing a new system certainly induces unwanted costs. The situation can be even worse in a commercial and competitive context, where this may result in customers abandoning the system and replacing it with a competitor that is better adapted.

It is disappointing to see that such a valuable principle as publish and subscribe does not help much to accommodate the potential needs of users. It is a fact that if the designer of a system has not provided the right interface, it will not be possible to access the information needed, even if it is buried somewhere in the system. It will also not be possible to get event notifications, unless some kind of subscription is available in the interface. For this purpose, the designer will have to choose which information is observable and when and how notifications will be propagated to users. With this scheme some extensions are possible, but sooner or later limitations will appear for instance because interesting events are not propagated or because the propagation scheme is difficult to use, or inefficient.

Some widely used systems have even been “victims of their own success”: because lots of users with conflicting needs were using them, they could not be further upgraded without “forking” into incompatible products. It is the same phenomenon that occurs with successful protocols like IP, HTTP or HTML: they are installed on millions of hosts, they quickly generate new needs for which new solutions exist but are almost impossible to adopt because of the costs of upgrading every interconnected host and because of the risk to introduce incompatibilities.

In the NAP report, eleven reasons why large-scale information systems are so difficult to design, build and operate have been identified and described. One of these reasons matches remarkably well our definition of evolution, thus, even if it wasn’t listed in the keywords defining a large-scale system, it is undoubtedly an important characteristic:

> *Constantly changing needs of the users—Many large systems are long-lived, meaning they must be modified while preserving some of their own capabilities and within the constraints of the performance of individual components. Development cycles can be so long that requirements change before systems are even deployed. Research is needed to develop ways of building extendable systems that can accommodate change.*
>
> (Committee on Information Technology Research in a Competitive World, 2000)

1.5 Distribution

Distribution aggravates the situation in several ways. We can distinguish **administrative** distribution and **physical** distribution, because both are present in large-scale IT systems and they introduce different technical aspects.

1.5.1 Administrative distribution

Administrative distribution reflects the fact that large distributed systems belong to different entities (people, organizations). For each information or functionality, some of these entities can be seen as providers, while others are rather users or consumers. Ideally, a provider must be able to satisfy the ever changing, sometimes contradictory, requirements of its users. This implies good communication and collaboration, as well as compatible objectives.

Collaboration problems may occur in many cases, for instance when there are so many users that the provider is unable to manage all their requests, or when users are too far, or when upgrading the service is too expensive or when all its developers are gone. In all these cases, collaboration between providers and users is difficult, and integration becomes problematic, hence:

> *System integration and the establishment of information systems that provide an open environment built for change and evolution have become the critical elements of modern information system development.*
>
> (Mowbray and Ruh, 1997, p. 215)

To achieve very large networks of inter-dependent sub-systems, we need to find solutions to keep them working in useful ways without the need for too much collaboration.

Another consequence of administrative distribution is that the owners of the different parts may have **conflicting objectives** or incompatible cultures, even if they are willing and able to collaborate. A striking example is provided in the context of CERN high-energy physics accelerators, where there is at least a dozen of groups involved in the engineering and operation of a single particle detector. Each group is highly specialized in one domain (construction, design, calibration, simulation...) and works with its own applications, database models and software, query dialects, etc. Although their ability to integrate their sub-systems is essential for the overall enterprise, each domain requires very specific features and unification of the models or applications is unthinkable because each of them is highly optimized for its own task. It is recognized that the inefficiencies that would result from an attempt to generalize the tools or models used could prevent the groups to accomplish their primary task.

A less dramatic example, that shows equally well the problems when subsystems belong to different organizational entities, is provided by hyperlinks between two Web sites. Very often, such links embedded in Web pages are broken and information becomes unreachable, just because of a simple rearrangement of the information at the target site. Such rearrangements are unfortunately required rather frequently, either because of aesthetic motivations, or for changes in the contents, or by the choice of a new underlying technology.

So we see that it is technically possible to establish “bridges” between independent domains (in the form of wrappers or hyperlinks) but that such dependencies are very brittle, and that each modification on one side potentially induces changes on the other side. The amount of work required to perform these changes can become prohibitive, especially for sub-systems that depend on many external sources. For instance the owner of a Web site with hyperlinks to 20 other sites that get reorganized every 6 months has already 40 updates to perform each year! Even if he can easily edit the contents of his own pages (no compilation or lengthy transfer) it still takes some time to understand the new organization of the information providing site and to locate the target page.

To avoid this tedious work, the only solution is to have an interface for each sub-system that is as stable as possible. Unfortunately, across organizations, the reasons to change a sub-system in incompatible ways may be much stronger than the willingness to preserve a stable interface. Another factor is that across organization it is often neither possible to know how the interface is actually used, nor to estimate the magnitude of the consequences of changes. The sad effects are seen much too often: unexpected loss of functionality, and inacessible or wrong information.

The bad effects of administrative distribution on the reliability of large systems is also mentioned in the NAP report:

> *Large numbers of individuals involved in design and operation— When browsing the Internet, a user may interact with thousands of computers and hundreds of different software components, all designed by independent teams of designers. For that browsing to work, all of these designs must work sufficiently well without anyone doing the integrating or anyone handling complaints if they fail to work as a whole. Research is needed on ways to prevent failures in one part of a system from affecting the system as a whole in ways evident to a user.*
>
> (Committee on Information Technology Research in a Competitive World, 2000)

1.5.2 Physical distribution

Physical distribution is another complicating factor whose technical implications on software development are well known (Waldo et al., 1994):

1. **Latency**: operations that involve the network are 4 to 5 orders of magnitude slower than local interactions. Ignoring this issue can lead to an application with extremely poor overall performances and response times.
2. **Memory access**: it is not possible to use pointers when objects are physically distributed, thus programming a distributed application either requires to distinguish local and remote objects, or to suppress pointers altogether (as in the Java language).
3. **Partial failure** is a tricky problem, emblematic of physical distribution, because some devices may crash while other continue running and there is no global state that can be examined to determine exactly what has occurred; building robust applications in this context requires astute techniques for exception handling.
4. **Concurrency**: having several devices that run in parallel introduces true concurrency and asynchrony, and unlike in the pseudo-parallel or multi-threaded context, it is not possible to rely on the deterministic behavior of a single operating system.

In their argument against location transparency, and against the idea to make local and remote objects undistinguishable for the programmer, Waldo *et al* give two very convincing examples that the non-functional aspects like **coordination** or **exception handling** are intricately linked with functional aspects—producing the expected result—and that they cannot be treated independently. They also warn the designers of distributed systems that the non-functional aspects may seem secondary when the size of the system is small but that they become essential when the load and the size of the system increase, and that handling them at this time requires major changes to the application.

Physical distribution is also likely to introduce **paradoxical effects**. Even if the interfaces of the components remain unchanged, there is a bigger risk that minor changes modify non-functional aspects (quality of service) in unexpected ways. For instance performing a linear search may give excellent response times for a small data structure, but it becomes unacceptable when more memory is added to a server and the data structure is able to grow. In this case an improvement (adding memory) results in degradation of service quality.

1.6 Messengers and mobile code

The messenger paradigm (Tschudin, 1993), on which most of our work is grounded, was introduced to suppress the need for pre-installed protocols in computer communications, by letting one of the communicating entities start a communication and give the description of the protocol to be used by the other party. Instead of exchanging messages, distant processes interact by sending programs (messengers) across the network. This is similar to the idea of "protocol components" that is used in active networking (Tennenhouse et al., 1997), in contrast to "protocol stacks" used by standard communication systems. This paradigm of messengers or mobile code is characterized by:

> The ability to send a program to a remote host, to have it executed there in a new thread of control, which is able to initiate further code transfers and thread executions from its remote location.

The exchange of mobile code allows the implementation of the same protocols as the classical message exchange, with the significant advantage that the code implementing new protocols can also be transmitted if necessary.

The result is that less conventions are needed in messenger exchange than in message exchange. Processes still have to agree on high-level encoding and synchronization primitives, but these agreements are only a fraction of what is necessary to communicate. We will show that many context-dependent aspects can be encapsulated inside mobile code and changed when the context changes. Encapsulation has the same benefits here as in other software engineering domains: it reduces the dependency between components, thus it reduces the number of modifications that we must make to our software in order to adapt it to new requirements. This crucial fact gives us new perspectives for the structuring of computer communications, and of distributed applications.

Code mobility has enjoyed a lot of popularity during the last decade. In an early paper (Harrison et al., 1995) the advantages of code mobility had already been described, and the fact that there are very few cases where a problem cannot be solved without mobile code was recognized. Afterwards, most of research on the topic attempted to build mobile code platforms, which requires a lot of efforts and advanced techniques to provide an acceptable level of security. A few interesting studies showed significant reduction of latency and consumption of network bandwidth, but the number of large applications described in the literature that could help someone build his own distributed application with mobile code is very limited.

Because deciding to use mobile code is not sufficient to successfully build a large distributed application, this dissertation describes the architectural principles that we applied to build a real-world service. It also tries to summarize our experience of applying mobile code to cope with some of the non-functional aspects that are typically found in DEDIS. Hopefully, dealing with these aspects can lead to more robustness, more flexibility, better accessibility and fewer discrepancies in these systems.

One last quote from the NAP report, that has already been cited several times in this introduction, indicates that our hope to improve the design of IT systems using code mobility is also shared by some of its authors:

> *Existing approaches to large-scale system design, including some that are in commercial practice, show promise for facilitating the development of large-scale systems and could benefit from greater attention from the research community. Two approaches worth mentioning are methodologies based on component software and mobile code.*
>
> (Committee on Information Technology Research in a Competitive World, 2000)

Chapter 2

The aspects of software design

Chapter highlights

- The motivation to build a catalog of aspects: as a theoretical tool to explain the difficulty of DEDIS and the impact of mobile code, and as a practical "checklist" for software designers.

- The definition of functional aspects: compute, store, communicate, the three basic operations that a Turing machine is able to perform.

- The notion of non-functional aspects: all other aspects that must be considered to obtain programs that fulfill their task in their environment.

- The implementation of non-functional aspects: even if they can be described in abstract terms and modularized, they necessarily result in more computations, storage and communications in the executable code.

- The complexity of aspect dependencies: because they are similar and tend to be solved by the same mechanisms, or because a given aspect introduces its own accompanying aspects, they are all interconnected in a very dense and complex graph of dependencies that make the design of software a challenging task.

- The distinction between aspects and quality factors: aspects are concrete technical concerns, and not desirable properties of the software.

- The problem of cross-cutting: it is almost impossible to cleanly modularize all aspects at the same time, and to avoid that code written to accomodate one aspect spreads within modules written for other aspects.

- The inventory of 38 non-functional aspects: attempts to define all of them and spans many fields of computer science.

2.1 Motivation

Designing useful software implies taking into account a large number of very concrete technical concerns, also named aspects. The goal of the present chapter is to provide a comprehensive overview of these technical aspects, in order to have a clear definition for each of them, and to know exactly which ones are taken into account in our architecture based on mobile code.

There is already an abundant literature documenting the particularities of each aspect, or of a few related aspects, and the best techniques or algorithms to take them into account in software. Surprisingly, we could not find any publication that tries to build an extensive catalog of all the non-functional aspects. Our motivation to provide such a document is twofold:

1. With it we can represent, and somehow understand, the enormous complexity of DEDIS, in which most of the non-functional aspects must be considered. For software designers, this makes a big difference with other kinds of applications (word processors, compilers,...) where only a small number of non-functional aspects are present and where the functional side predominates.
2. It could also provide useful guidance to software designers, by ensuring that they are aware of all the aspects that are relevant to their problem, and to remind them of the complex dependencies between them.

2.2 Functional and non-functional aspects

As **functional aspects**, and because their behavior can be compared with mathematical functions, we designate **processing**, **storing** and **communicating** information: the three fundamental tasks accomplished by all computers according to von Neumann's architecture. These are also the operations of the the underlying theoretical Turing machine (an abstraction that is very close to a mathematical function), of "functional" programming, and of some formalisms that allow the mapping of programming constructs with predicates, in order to verify their correctness (Dijkstra and Scholten, 1990).

So, when we describe a system's **function**, we are describing its behavior along these three axes, and we're viewing it as a component that has a state (information stored in its memory), and that is able to communicate with the world using input and output channels. The operations that read the input and compute the output may take the state into account and modify it. Ensuring that a system's function meets expected requirements is a very difficult task and has been the subject of numerous formal methods, e.g. Z (Spivey, 1992) or CO-OPN/2 (Biberstein, 1997).

Non-functional aspects on the other hand are additional requirements that **must** be taken into account in some cases. Thus it is possible to design software with the right functional behavior but that is not correct because it fails to address these needs. For instance a program that always computes the right result but sometimes takes too long doesn't satisfy the aspect of **real-time constraints**. Another easy example is **memory management**: a program may require more memory than available in the host it's running on. If the program's designer has not taken the aspect into account it will fail even if the algorithm is perfectly correct.

Even if such problems may seem secondary, and have sometimes been described as "quality of service" or "implementation issues", they do have a heavy impact on the actual programs written to provide the desired function in a given context. Indeed, handling the specific problems introduced by the non-functional aspects often requires to incorporate more software in the system, and may also constrain the applicable algorithms or their implementations.

Many of the aspects presented have the characteristic of requiring computations, storage and communication for their own purposes, or to introduce additional aspects in a kind of cascading phenomenon. For instance, the aspect of **load balancing** between several processing nodes requires rather frequent communications between the nodes to gather load statistics, it also requires some memory to store this information, and it requires computations to determine where the processes should execute. It is unlikely that separate processors, memory and network connections will be available to perform this task, thus load balancing consumes resources and is very likely to interfere with the primary function of the system.

Such examples illustrate the fact that the three fundamental aspects and what we conveniently call non-functional aspects are never very far from each other. Thus, we will cautiously avoid the notion of orthogonality, that has sometimes been introduced in this context. On the contrary, we observe that there is a very complex network of dependencies between aspects, either because they are similar and tend to be solved by the same mechanisms, or because a given aspect introduces its own accompanying aspects. Fig. 2.1, p. 37 depicts this network of dependencies.

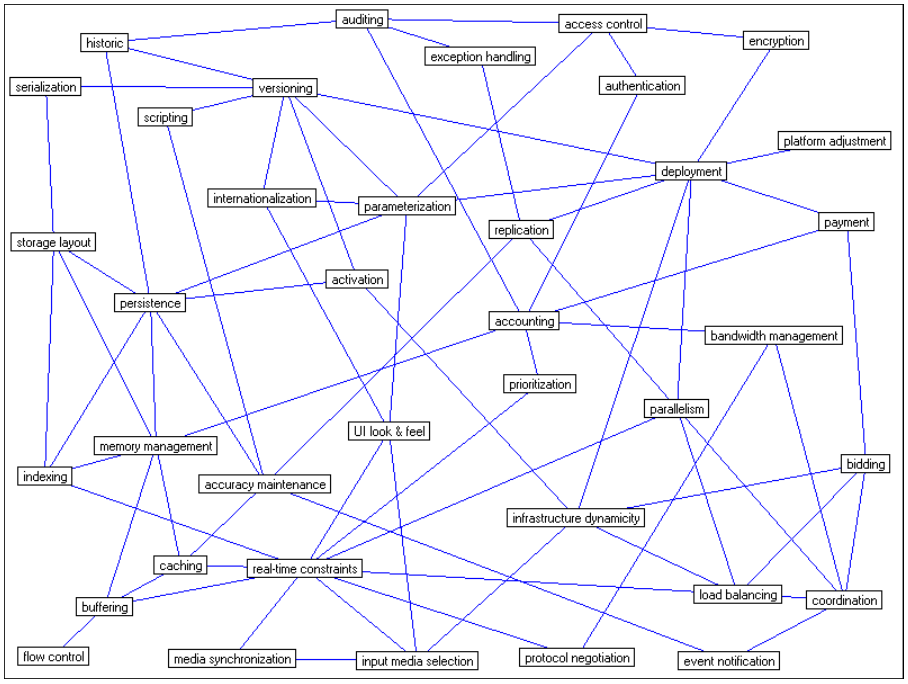


Figure 2.1: Dependencies between non-functional aspects that have been identified and described in section 2.5.

2.3 Design goals and quality factors

It is important to notice the difference between **technical** aspects and more abstract goals and quality factors such as: reliability, robustness, maintainability, modularity, scalability, adaptability, portability, autonomy, data accessibility, inter-operability, appliance connectivity, transparency, security... (Tiemann et al., 1997). Although such goals may have a big influence on the choice of a technology, or on the strategy of an IT department, they are not technical, in the sense that it's much less clear how to solve them programmatically. These quality factors may or may not be present in the software, depending on how well the technical aspects were handled and integrated in the design, but they don't have the same kind of direct impact on the software written.

2.4 Cross-cutting

Several authors (Lea, 1997; Kiczales et al., 1997) have pointed out that most difficulties in programming arise from the complex inter-relations between these aspects. These complex inter-relations engender the problem of cross-cutting, which compromises our ability to design independent modules or components. While modular decomposition has a crucial impact on reuse, ease of design and maintenance of software, it is often not possible to find a decomposition that accomodates all aspects at the same time. In other words, when one modular decomposition is chosen, it is difficult to keep the code written to accomodate other aspects in one single place, without having to spread it over all modules.

Fortunately, there are some techniques, like aspect oriented programming (Kiczales et al., 1997), design patterns (Gamma et al., 1995) or "service encapsulation" (Lea, 1997) that can help the software designer coping with this cross-cutting problem.

2.5 Catalog of non-functional aspects

For this inventory, we chose to list all aspects in alphabetical order. This presentation allows a given aspect to be quickly located, but somehow also reflects our current incapacity to find a better classifying criteria. In Fig. 2.2, p. 39, we propose an overview of all aspects, roughly structured in seven categories.

It was easier to structure the presentation of each aspect. Thus, all aspects descriptions follow the same model, made of four distinct parts:

1. **When** part: indicates in which circumstances the aspect is likely to be found.
2. **Consider** part: describes the main technical issues raised by the aspect.
3. **Techniques** part: references well-know techniques used to implement the given aspect.
4. **Related to** part: lists all related aspects and describes the relationships between them. The relationships presented correspond to the links in Fig. 2.1, p. 37.

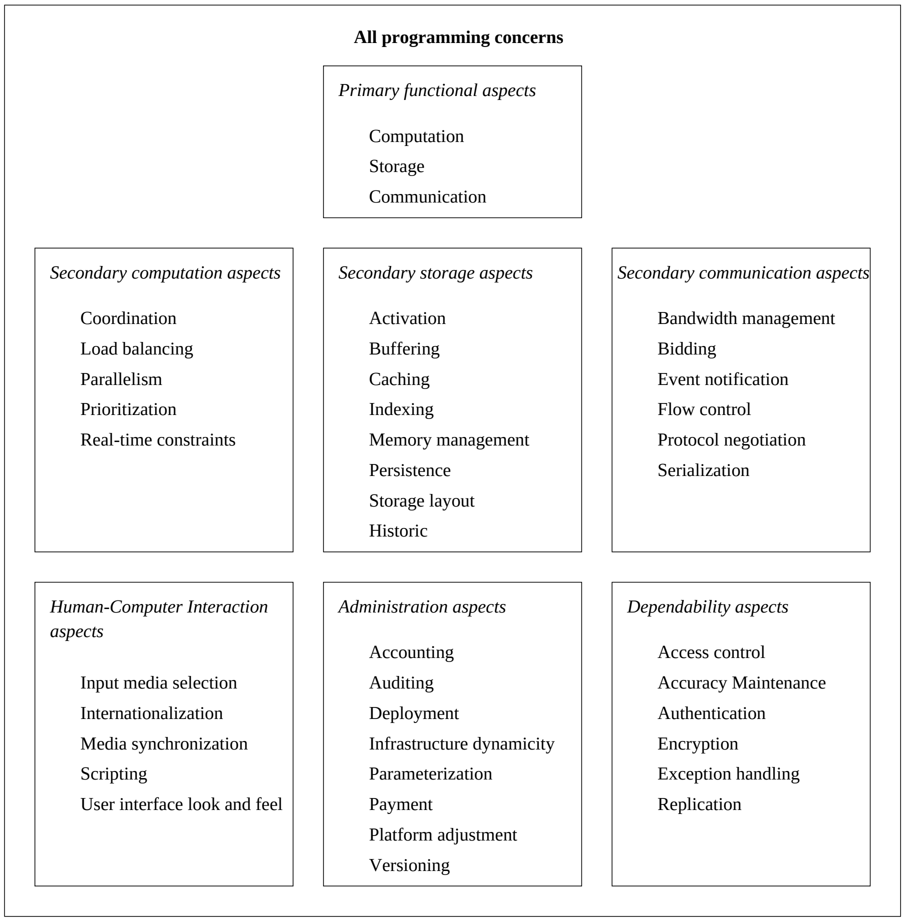


Figure 2.2: An attempt to categorize aspects. Unfortunately, most aspects do not fit well in a single category. The difficulty to determine clear categories can also be perceived in the very dense network of dependencies visible in Fig. 2.1, p. 37.

1. **access control**

When	some data or operations should not be available to all actors interacting with the system; for safety or privacy reasons
Consider	defining a reference monitor that controls attempts made by subjects (principals) to access objects (data, operations)
Techniques	access control lists, capabilities
Related to	**auditing** to detect attacks and violations, **authentication** to identify subjects, **encryption** to hide objects, **parameterization** to let administrators configure the reference monitor

2. **accounting**

When	the consumption of resources (CPU time, bandwidth, volatile and persistent memory, software components) must be known
Consider	tracking the principals owning threads, needing memory or using the network infrastructure
Techniques	run time monitoring at low level
Related to	**auditing** to report actual usage, **authentication** to identify principals, **bandwidth management** requires accounting, **memory management** requires accounting, **payment** can be based on resource usages, **prioritization** requires accounting

3. **accuracy maintenance**

When	the system's state is related to a dynamic environment where various kinds of events can occur
Consider	how to update the system's internal state such that it continuously represents the environment in an accurate way
Techniques	human intervention, sensors, polling
Related to	**caching** and **replication** require to maintain coherence between the original and the replica, **event notification** is necessary to maintain accuracy in external components, **scripting** is useful to propagate changes automatically, **persistence** requires that stored data is maintained up to date

4. **activation**

When	stopping and restarting the system may cause a loss of state information or has an undesirable impact on some operations (changing their behavior while it should remain the same; or the opposite)
Consider	how to preserve the state and the right behavior, when the system is restarted, when an operation is invoked for the first time, when a user reconnects, etc.
Techniques	object-oriented activation frameworks, transfer of state to another host that won't be stopped
Related to	**infrastructure dynamicity** (e.g. crashes) causes systems to stop and restart, **persistence** of entity (database) and session objects is necessary, **versioning** may require stopping and restarting systems

5.

auditing	
When	the system's behavior is expected to meet some well-defined objectives
Consider	which observations are necessary to determine that the objectives are met
Techniques	methodical examination of the execution, logs
Related to	**accounting** requires observations, **access control** should be verified, **exception handling** must report problems, **historic** records changing values over time

6.

authentication	
When	the identity of some principals must be determined
Consider	how this data will be inserted, stored, checked, used, and deleted
Techniques	passwords, cryptography, biometry
Related to	**access control** requires identification of principals, **accounting** may require indentification of components

7.

bandwidth management	
When	several applications compete for bounded network resources
Consider	that "worst-case" static allocation will waste resources when some applications are idle; better resource utilization can be achieved by "over-booking" and designing applications that are still able to work under non-optimal conditions
Techniques	"best effort", schedules, multi-level encoding
Related to	**accounting** of bandwidth usage is required for good management, **coordination** of communications to avoid demand peaks helps achieving good bandwidth usage, **protocol negotiation** may favour approaches consuming less bandwidth

8.

bidding	
When	choosing between several options is a matter of trade-off (e.g. which resource should perform a given task)
Consider	that an overall good behavior can be achieved by letting each component behave rationally, trying to maximize a given utility function
Techniques	contract-net protocol, auctions, leveled commitment
Related to	**infrastructure dynamicity** and **load balancing** require choices at run time, **payment** can be related to the bids, **coordination** of tasks may result in better resource usage when needs are known in advance

9. **buffering**

When	input data arrives in bursts, or synchronous communication is not convenient, or performance suffers from disparities in the size of data handled at both ends of a communication channel
Consider	storing data such that the receiver can handle it asynchronously and at its own tempo
Techniques	temporary storage, queues
Related to	**caching** is closely related but is more an issue of performance, **flow control** becomes necessary when buffers are not sufficient, **memory management** becomes an issue when buffers are used, **real-time constraints** can motivate the use of buffers

10. **caching**

When	a relatively large amount of time is needed to obtain or save the result of an operation (computation or communication)
Consider	storing a copy of this result in a quickly accessible memory
Techniques	ensure that changes are propagated in time
Related to	**accuracy maintenance** becomes important when data is replicated, **buffering** is closely related but is more an issue of synchronicity, **memory management** becomes an issue when caches are created, **real-time constraints** can motivate the use of caches

11. **coordination**

When	there are dependencies beteween several activities, like precedence constraints or conflicts through shared resources (concurrency)
Consider	the nature of these dependencies and how to manage them; the fact that deadlocks may occur when tasks require more than one critical resource non atomically or in changing order
Techniques	clock synchronisation, scheduling and constraint satisfaction when the activities must be accomplished within known time bounds, or in a given order; data driven (focus on interactions) and control driven (focus on system configuration) coordination languages; concurrency control with semaphores, deadlock prevention algorithms, timeouts
Related to	**event notification** is a way to deal with "prerequisite" dependencies, **load balancing** and **bidding** try to deal with irregularities when there are dependencies through shared resources and demand is not know in advance, **bandwidth management** can be made easier by coordinating the needs, if they are known in advance, **parallelism** introduces concurrency problems

12.

deployment	
When	the software is not running at a single location
Consider	where it will be installed, how and by whom
Techniques	desktop application, applet, client/server, embarked
Related to	**encryption** can be necessary to protect the software during deployment, **infrastructure dynamicity** requires deployment when new devices become available, **versioning** must be considered when deployment occurs several times, **parallelism** and **replication** may require deployment if there is no shared storage **parameterization** of software running at several locations must be considered from a global perspective, **payment** of deployed software can be required, **platform adjustment** is necessary when the devices or environments are not homogeneous

13.

encryption	
When	security sensitive data must be hidden
Consider	why and how it will be encrypted, who (which "principal") will receive the keys to access it, when, how and for how long
Techniques	cryptographic algorithms, keystores, certificate authorities
Related to	**access control** requires data to be hidden, **deployment** may require encrypted software

14.

event notification	
When	events occuring in one component must be observed by other components
Consider	which are these events, how can they be propagated in a safe and efficient way
Techniques	publish and subscribe, event queues
Related to	**accuracy maintenance** requires that events are visible outside of the components, **coordination** of two activities linked by a "prerequisite" constraint needs an event notification when the first activity terminates

15.

exception handling	
When	a component behaves in an unexpected way
Consider	how the failure can be contained such that it does not spread to other components and "contaminates" the whole system
Techniques	"ACID" transactions to clean up state after failures (checkpointing), dump and halt
Related to	**auditing** is necessary to understand exceptions and correct faults, **replication** increases system availability in presence of exceptions

16. **flow control**

When	a data source works at a much higer byte rate than a data consumer
Consider	how to adapt the source's output rate to the consumer's ability
Techniques	in-band/out-of-band signals, request/reply, sliding window
Related to	**buffering** can be a way to avoid complex flow control

17. **historic**

When	it is necessary to remember the values of objects at the time a business transaction took place
Consider	which values may change over time, how often they change, and if it is useful to be able to do an analysis of the change over time
Techniques	snapshot values in the transactions if there are not many transactions between value changes (too many duplications) and if doing an analysis is not required (bad sampling); add classes to record changes and return the value at a given time, behind the objects that must be recorded
Related to	**versioning** is more about the successive versions of the programs, but it may constitute important historic data too, **persistence** of the values is necessary, **auditing** also requires logs and historic data but for administrative purposes

18. **indexing**

When	the system handles a large amount of data
Consider	providing efficient means to locate this data
Techniques	database keys, search based on content or properties, batch or immediate index updating
Related to	**memory management** can be an issue for very large indexes, **persistence** of lots of data may require indexing, **real-time constraints** motivate the creation of indexes, **storage layout** must be taken into account, both for the indexes and for the data being indexed

19. **infrastructure dynamicity**

When	components or devices may be added to, removed from, or moved in the system at any time
Consider	how other entities in the system can discover their addresses and characteristics in order to establish a communication and collaborate with them, how this communication can be suspended and re-established; where data and computations must be located to deliver expected results
Techniques	name servers, directories, broadcast, lookup service, routing tables, discovery
Related to	**activation** becomes important in a dynamic environment, **bidding** and **load balancing** techniques can be applied to cope with dynamicity, **deployment** of components cannot be static, **input media selection** may be restrained in very dynamic settings

20.

input media selection	
When	the system's input may be provided in several ways
Consider	data quality, ease of use, consistence within the system and with the environment, reliability
Techniques	keystroke data entry, automatic scanning, voice entry systems, touch tone, bar code
Related to	**infrastructure dynamicity** (e.g. frequent disconnections) or **real-time constraints** can dictate the best choice, **media synchronization** issues can arise, **user interface look and feel** is more about presentation

21.

internationalization	
When	the system is meant to be used by people speaking different languages
Consider	providing interfaces in their native language; and adapt to the internationalized version of the OS
Techniques	manual/automatic widget relabeling, resource database
Related to	**parameterization** is one way to tackle language issues, **user interface look and feel** can be influenced by the language or culture, **versioning** is another way to tackle language issues

22.

load balancing	
When	using several processors in parallel will not yield the expected speedup because the problem is irregular and some processors take much longer than others to complete their part of the work
Consider	adapting the task of each processor frequently, to obtain even processing times
Techniques	task pooling, domain adjustment
Related to	**bidding** can be used to adjust loads, **infrastructure dynamicity** can cause loss of performance or new opportunities, **parallelism** is the context in which load balancing becomes necessary, **real-time constraints** may require a high level of performance, **coordination** of the activities may solve the same problems when the capacities and loads are known in advance

23.

media synchronization	
When	information is conveyed by different media strands
Consider	that these different media must be synchronized at the receiver, e.g. that lips in video move with speech output
Techniques	timestamps, common clocks
Related to	**input media selection** is more a matter of ergonomics but can be constrained by synchronization aspects, **real-time constraints** are introduced by this aspect

24. **memory management**

When	the system's needs may exceed available memory
Consider	how much memory is available, how much will be needed to represent the system's state and to perform the various operations, when data may be removed from an overloaded memory
Techniques	compression, swapping to secondary storage (virtual memory), garbage collection, leases
Related to	**accounting** of memory usage is required for good management, **buffering**, **caching** and **indexing** must achieve a good trade-off between performance and memory usage, **persistence** uses secondary storage that must be managed, **storage layout** may impact the amount of memory needed

25. **parallelism**

When	a single processor is not able to achieve sufficient performance, e.g. to handle a very high number of requests or to carry a complex computation within narrow time bounds
Consider	using several processors running in parallel
Techniques	shared memory, cluster of workstations
Related to	**coordination** may become necessary when several processors work in parallel, **deployment** of software on several nodes may require more effort, **load balancing** may be needed to achieve good performance, **real-time constraints** motivate the use of parallel processors, **replication** of hardware is similar to parallelism, but for the purpose of fault-tolerance, thus replicated devices perform the same task

26. **parameterization**

When	the system's behavior can be modified by an administrator or end user
Consider	the scope of the modification: how it will affect the system's behavior for other users, how the change can "propagate" to other computers used by the same person
Techniques	property files, user profiles
Related to	**access control** must be parameterized, **deployment** of software at several locations may require global parameterization, **user interface look and feel** and **internationalization** often require parameterization, **persistence** of parameters must be considered, **versioning** should take into account existing parameters

27. **payment**

When	the system is not completely available for free
Consider	who will pay, how much, for what, by which means; and how to design the system in order to prevent abuse
Techniques	payment in real or fictitious money when downloading or installing software, accessing a service, consuming resources
Related to	**accounting** in a "pay per use" setting, **bidding** when the transactions have a mapping to real money, **deployment** when having the software on one's computer requires paying a fee

28. **persistence**

When	part of the system's state must be preserved after it is stopped
Consider	when and how this data is created, destroyed, updated and recalled
Techniques	non-volatile storage, databases
Related to	**accuracy maintenance** is necessary when data is stored on a persistent medium, **activation**, **historic** require persistence, **indexing** can be useful with lots of data, **memory management** is required also with persistent media, **parameterization** requires to store parameters, **storage layout** is more about the structuration of stored data

29. **platform adjustment**

When	software is meant to run on several platforms
Consider	what the relevant differences will be regarding the operating system (disk, graphics, security), hardware compatibility, resources (memory), performance
Techniques	virtual machine, least common denominator, platform-specific code
Related to	**deployment** in an heterogeneous environment

30. **prioritization**

When	some tasks require more processing than others, in order to be completed in time
Consider	assigning a higher priority to these tasks and biasing the selection process towards higher priorities; ensuring fair sharing and avoiding livelock may be difficult
Techniques	scheduling policies (static/dynamic priority, fixed/dynamic time slices, lottery scheduling, sporadic server execution)
Related to	**accounting** of actual resource usage is necessary, **real-time constraints** induce this aspect

31. **protocol negotiation**

When	several protocols, or protocol parameters are possible for a given communication, with various resource requirements, and different safety or privacy properties
Consider	which protocol is the best one for the given situation and how communicating parties will come to an agreement on these attributes before the conversation begins, to guarantee its soundness
Techniques	phases or levels in communication
Related to	**bandwidth management** is about available network resources, **real-time constraints** may influence the choice of a protocol

32.

real-time constraints	
When	the correctness of the operations depends upon the time at which the result is produced, or the amount of resources consumed
Consider	counting or measuring the resources they need and ensuring that their execution is not delayed by other operations or blocked because a resource is not available
Techniques	schedulability analysis, Rate-Monotonic Analysis
Related to	many aspects that impact performance: **buffering**, **caching**, **indexing**, **load balancing**, **parallelism**, **prioritization**, **protocol negotiation**; on the other hand, several aspects which introduce new real-time constraints: **input media selection**, **media synchronization**, **user interface look and feel**

33.

replication	
When	failures that may cause loss of data or service interruption are not acceptable
Consider	duplicating capabilities or components that may fail (disks, computers, network) for backup or continuity of operations; this will also incur costs and new consistency maintenance problems
Techniques	duplication of storage and communications devices
Related to	**accuracy maintenance** becomes more complex when redundancy is added, **deployment** on replicas must be considered if there is no shared storage, **exception handling** also tries to deal with failures, **parallelism** replicates devices to improve performance but lets them perform different tasks

34.

scripting	
When	the system is used for routine work
Consider	alleviating the user's work or avoiding mistakes in repetitive tasks with recorded scripts
Techniques	macro recording, tool command languages
Related to	**accuracy maintenance** may need scripts to update data, **versioning** must preserve existing scripts

35.

serialization	
When	data is transported or made persistent
Consider	how this data will be represented for transport and storage, such that the nature of this data (type, version) is preserved, as well as the structure and the relationships with other data (objects, object graphs)
Techniques	TLV, ASN.1, XML
Related to	**storage layout** implies transformation of data, **versioning** must take into account existing serialized data

36. **storage layout**

When	the system stores complex data
Consider	the logical structure of this data, define a place for each piece of data
Techniques	objects, arrays, records, files, relational tables, XML pages
Related to	**indexing** can be used for better performance but is depends on the layout, **memory management** may constrain the choice of a layout, **persistence** is about choosing which data must be stored, **serialization** is about representations, not necessarily for storage

37. **user interface look and feel**

When	an interactive system has a complex user interface (possibly graphical)
Consider	identifying user categories, following good human factors engineering guidelines, identifying a set of allowable widgets and combinations of widgets, providing a consistent look and feel for the project
Techniques	dialogs, wizards, online help, Web pages
Related to	**input media selection** also has an impact on ergonomics, **internationalization** makes this aspect more complex, **parameterization** can improve comfort of the user, **real-time constraints** express the need for short response times

38. **versioning**

When	new versions of the system with additional features or corrections are deployed
Consider	preserving user's data, remaining compatible with previous versions, preserving user's working habits; for centralized services downtime must be minimized and session/transaction semantic must be preserved
Techniques	data conversion, adapters, service replication
Related to	**activation** can be required when the system has to be restarted to take corrections into account, **deployment** may require to maintain several versions or to propagate updates to several locations, **internationalization** may require several versions, **parameterization**, **scripting** and **serialization** are related because serialized information, scripts and parameters must remain valid with new versions, **historic** records changing values over time, version changes sometimes need to be recorded

Chapter summary

At the time of this writing, this list seems to be the first attempt to describe all technical aspects of software and their relationships. Although most of them have been studied for a long time, the need to understand how these concerns interact and how they can be handled separately in software has been recognized only recently.

Chapter 3

Service architecture

Chapter highlights

- The shift from application to services: because computers handle an increasing part of information and processes, and because network connections are generally available, being able to integrate and adapt software has become a crucial economical necessity. Instead of stand-alone applications, software should be designed as services: modularized parts of large distributed systems, that can easily be integrated into new software.

- The shortcomings of current techniques: various approaches hope to enable the construction and integration of services, but they are limited because they don't recognize the importance of dealing with non-functional aspects, or because they rely on too many rigid conventions.

- The approach we recommend: using mobile code for the integration of immobile services. It offers (1) simpler service design, (2) easier integration and adaptability to new functional and non-functional requirements, without requiring collaboration between service providers and clients.

- The other standpoints on mobile code: it has some quantitative advantages, but also some dark areas that may have hindered adoption until now.

- The illustration of protocol encapsulation: what is this fundamental difference between message exchange and program exchange; and how it will help us manage change by reducing the number of necessary conventions and standards.

- The architecture of services: the main components of a service that runs within an execution environment; the roles of these components with respect to the functional and non-functional aspects; and detailed technical descriptions of these components to help service designers that would like to follow our approach.

- The required features of the execution environment: what support the architecture expects from the environment, in addition to the ability to move programs.

3.1 From applications to services

We have seen that in the recent years the nature of information technology has changed and that the number of systems that show a pressing need for flexibility and easy integration has become significant. This is not surprising since in our society the computer is replacing ink and paper for the storage and the diffusion of an increasing part of knowledge, tasks, and transactions. Once data and processes are managed by computers, and networking these computers is feasible, there can be very strong incentives to access and use these assets from outside of the original system.

In the context of DEDIS, the potential ways to use an information system or business application are completely different from what had been made in the previous decades (stand-alone applications on personal computers or departmental applications on mainframes). Technical limitations like network bandwidth, processing power or storage are vanishing, at the same time the cost of computers and appliances is falling. Today, in Switzerland, WAP-enabled mobile phones are distributed for free to people signing for a 12 month subscription...

Nowadays, when building a new information system, it is not wise to aim only at fulfilling a single business need. Even if the goal and purpose of the system are well defined, and the economic motivations to develop it are clear (reducing operation costs, enabling a new business, etc.), there may be other advantages well beyond the initial problem solving goal, and it could be very regrettable not to exploit them. There is a clear necessity to adopt a different point of view and to consider systems not as closed, single purpose applications, but as **services** which are not only developed for a few people in a department but that may be offered to a wider audience. This is clearly reflected in the current tendency to promote "business-to-business" and "business-to-consumer" computerized relationships.

This shift from applications to services is required by the impossibility to know all possible uses in advances. For this purpose, the notion of service comes as a natural extension of the notion of server. The term server is rather used to describe a process running on a defined host and providing information to its clients (e.g. a Web server). Usually client and server communicate by exchanging messages according to a predefined protocol (e.g. HTTP). The notion of service is more abstract: it encompasses the notion of server, as well as other software entities that are not necessarily running on a well defined host and that are not interacting with their environment with a limited request/reply scheme (e.g. Web service, encryption service, naming service, routing service, etc.). It is a powerful notion because it allows the decomposition of large distributed applications into smaller elements that are easier to understand and reuse.

To understand the need for this shift from applications to services is a first step, but it is not sufficient to achieve reliable integration on a large scale and in a dynamic context:

> *While system software manufacturers have long recognized the need for tools that support system integration, many have fallen short of satisfying the needs of the system implementors. The tools that do exist consist primarily of scripting languages, code libraries and clipboard functions. Most of these provide only low-level communications-oriented functions, such as sockets and remote procedure calls. Rarely do these tools support integration across heterogeneous operating system platforms.*
>
> (Mowbray and Ruh, 1997, p. 7)

The authors of the previous quote tell us that the integration problems are not yet solved in a satisfactory way, and that it is necessary to find better ways to perform service integration. To support CORBA, their integration technology (§ 4.6, p. 85), they claim that the abstraction level in other technologies is not sufficient and that the job of the programmer can be made easier by an additional layer of abstraction that hides distribution.

To balance this optimism we must also take into account the controversial paper of Waldo *et al* which is **against** this kind of abstraction. For them, transparency is **not** the right way to improve the quality of information and the robustness of DEDIS. These authors convincingly argue that the problems don't result from difficulties in communication, but from the interference of what we call non-functional aspects. It is rather because of the high number of non-functional requirements, and the complex dependencies between them that DEDIS are difficult to build and not because existing tools are not making communication easy enough. Thus, tools that make communication even easier without improving the management of other aspects are likely to be useful only for small-scale projects.

> *The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with differences in memory access paradigms between local and distributed entities. People attempting to write distributed applications quickly discover that they are spending all of their efforts in these areas and not on the communications protocol programming interface.*
>
> (Waldo et al., 1994, p. 5)

This rejection of transparency is also present in (Tschudin, 1993, p. 135) because all communication networks actually have an internal structure which is almost impossible to hide completely. Again, the functional aspect may be well handled, but quality of service issues are out of control. Thus, any large enough distributed application will be disrupted by the network's internal nodes, their resource limitations, and other applications that compete for these resources. In this context, making the network's behavior palpable or even programmable is a great argument in favour of messenger based communications.

In any case, the fact that reputable authors can have such contradictory points of view is revelatory of a domain where certainties are extremely difficult to achieve, and in which the study of new approaches is necessary.

3.2 Composition of services with mobile code

A few studies have shown quantitative advantages of using mobile code instead of other approaches, for instance in the domains of document filtering (Johansen, 1998), of network management (Baldi and Picco, 1998), or in the domain of indexing (Brewington et al., 1999). In all these cases, moving code to perform computations where data is located allows better performances (shorter response times or lower bandwidth usage) than approaches based on message passing, even if the improvements can be observed only when the ratio between the size of data and the size of the code is above a certain threshold. Clearly, these quantitative benefits are not sufficient to trigger a large adoption of the new paradigm. In (Milojicic, 1999) at least five issues that may have hindered adoption are invoked:

- **security**: protecting a server from malicious agents and an agent from malicious servers are both hard problems;
- **programmability**: mobile agents are too difficult to program for end users, and even trained programmers may be scared off by new concepts or programming environments;
- **scalability**: no one has attempted to operate a network with millions of mobile agent platforms;
- **lack of killer application**: only a few applications have been described, and admittedly, they could have been programmed without mobile code technology;
- **lack of programming model**: the development of applications with mobile agents requires guidelines or a methodology that has not been described yet.

It is essentially on the last point that we want to elaborate. Clearly, some software components are too complex or too large to be moved (e.g. a large database), and some are physically bound to resources that have a definite location like screens or hard disks. But there are some cases where the constraints are not so clear and where it is more difficult to dismiss mobile code, or to make proper use of it.

We advocate the use of mobile code as a flexible solution for the integration of immobile services. The main reasons to use mobile code this way are practical and qualitative, as opposed to the quantitative studies on network bandwidth and latency mentioned above:

1. When a service is running within a mobile code environment, a few non-functional aspects can be **Delegated to the Client**. This shift of responsibility is made possible by mobile code, that brings the exact functionality needed by the client on the server host. Since he doesn't need to consider these aspects, the designer of the service can concentrate on its main features. In addition to simplify service design, this delegation offers adaptability of the service to the specific needs of clients, and guarantees integration with future applications, communication protocols and devices.

   Being able to delegate some aspects to the client doesn't mean that the service providers will necessarily do it and that customers will have to implement their own access routines. We can expect that service providers supports a standard set of clients, with a basic set of features, at least in order to test their programs. What really matter is that they don't have to take all possible cases into account, and that their systems will not be threatened by new market tendencies, user needs or technologies.

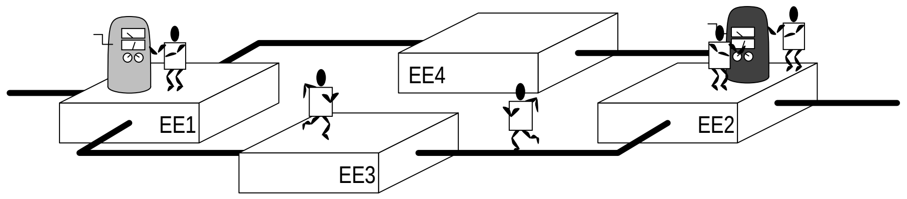


Figure 3.1: Two services (rounded gray machines with dials and buttons) interacting by means of mobile code and installed in a network of four interconnected execution environments (white boxes).

2. For several aspects, being able to execute code directly on the machine where the service is running offers a much better control than interacting with a remote host. In this case, the client **Benefits from Locality**, he is able to implement his own strategies, for instance to optimize resource usage, or to handle event at the source, even in disconnected and asynchronous conditions.

   These are not the same aspects, nor the same benefits that are visible from the provider's perspective. But it is important to note that clients also have additional possibilities, and that they have their own interest to work with a service that runs within a mobile code execution environment.

3. For another set of aspects, the mobile code **Execution Environment** provides adequate support, on which the service provider can rely. These are aspects which are sometimes not well supported by plain operating systems, and hence, the design of applications in the new context will be simpler even if mobility of code is not directly involved.

A detailed description of these three categories of benefits, based on the case study of Part II, will be presented in § 10.1, p. 191.

Hence, running services inside mobile code execution environments and using mobile code to perform the communications between them can have an extremely positive impact on their integration, and on the underlying quality factors like accessibility, reliability, etc. Fig. 3.1, p. 55 is a highly schematic depiction of this idea.

Let S1 designate the service installed on execution environment EE1 and S2 the service in EE2. In most classical distributed systems, which are based on message exchange, S2 is able to interact with S1 by sending requests and receiving replies according to a predefined protocol. The messages exchanged are named protocol data units (PDUs), and both S1 and S2 must be equipped with compatible software that is able to listen for incoming PDUs, decode them, then reencode and send the appropriate answer. PDU based communication, which has also been named client/server architecture or request/reply schemes exists in many variants and at many abstraction levels. Some well known examples are the Hypertext Transfer Protocol (HTTP) which is used by Web clients to request pages from Web servers, the Structured Query Language (SQL) which can be used to query and modify information in databases, or Remote Procedure Calls (RPC) for interaction between objects located in different execution environments.

The key additional elements in mobile-code based technologies are that one of the communicating components (e.g. S2) may

- install new programs on the other platform (EE1), in order to add functionality;
- spawn new threads of control on EE1;
- instantiate new objects (data structures) on EE1 in order to store informations that have been brought from EE2, or the results of local computations;
- invoke the methods of S1's API locally, from inside EE1, without crossing the network, and thus without suffering from possible latencies, disconnections or other non-functional aspects.

More detailed characteristics of mobile code execution environments, like the deployment of distributed applications, the means to obtain references to components, to coordinate tasks, to manage resources, to secure informations, etc. will be discussed in § 3.5, p. 68.

In summary, the essence our approach is that it tries to exploit the ability of mobile code to encapsulate protocols. This is motivated by the observation that DEDIS must be built with technologies that require as little collaboration as possible between service providers and users, and that provide as many opportunities as possible to evolve and to adapt to the dynamic nature of the networked infrastructure.

3.3 Protocol encapsulation with a mobile object

Protocol encapsulation is possible when a communicating entity (client or service provider) can provide some pieces of code to its interlocutor, to instruct it how to communicate.

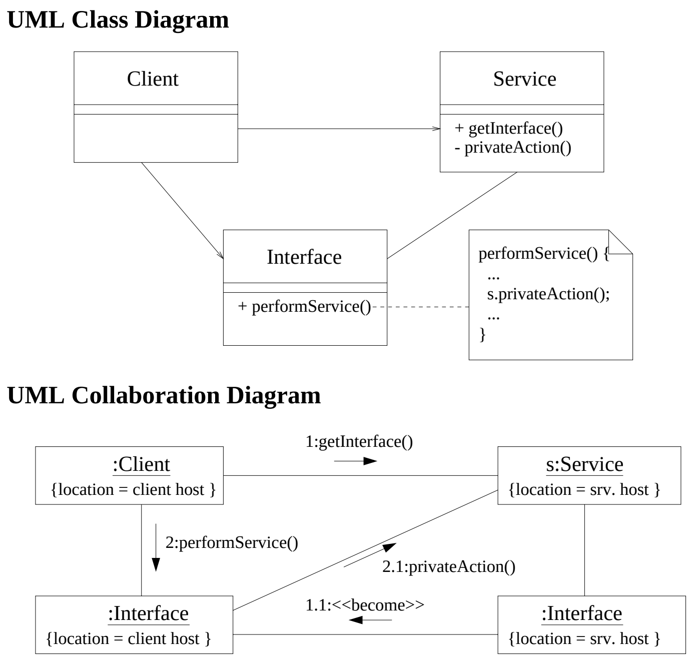


Figure 3.2: UML diagrams for protocol encapsulation.

Using UML object-oriented notation (Booch et al., 1998), we can illustrate this singular ability of messengers and other forms of code mobility (Fig. 3.2, p. 57). In this case, the mobile object goes from service to client (and encapsulates the protocol of the service) but the opposite is valid and useful as well. Notice that we have not represented some details pertaining to the language and model of code mobility (class loading, remote method invocation, type checking).

This example assumes that the service provider has a proprietary protocol for accessing his service. He must instruct each client how to communicate with it. Furthermore he wishes to hide the protocol in order to protect it or to be able to change it without informing the client.

Mobility provides an elegant solution to this problem. Only two conventions must be pre-established: (1) how the client retrieves the service interface (e.g. by calling the public method `getInterface`), and (2) which method of the interface the client must call to access the service (e.g. `performService`). It is important to notice that the code of the interface is not available on the client side before it has been requested and moved by the service (`<<become>>` stereotyped message) and that it is inside this code only that the service provider describes the private protocol (the `privateAction` method).

If we consider that no code mobility is used, all this example is similar to the (remote) proxy design pattern (Gamma et al., 1995), which also allows the protocol of an object to be hidden. We must not be mislead by the apparent simplicity of this proxy mechanism. It can be extremely cost-effective if the service is used by thousands of clients which do not need to preinstall anything else than a standard execution environment. In addition to being able to update his private protocol without disturbing a single client, the service provider is also able to differentiate clients, namely to provide different interfaces according to the needs of the clients, without changing his private protocol. It is this ability to customize protocols that makes mobile code an excellent tool to take into account the nonuniform and rapidly changing non-functional aspects related to communication and physical distribution.

Defining protocols is a tedious task and implementing them can be even worse. Thus every time that an organization is able to update software unilaterally, without collaboration, there is an interest in using mobile code, and DEDIS seem to provide numerous such occasions. We can illustrate this difficulty to achieve interworking, or to change a protocol once it is widely deployed with numerous examples, from different standards for electric plugs to the painful transition from IPv4 to IPv6. Here I want to report a simple but significant recent personal experience:

1. I receive an email with two PDF (Portable Document Format) files attached. Using Netscape on my Solaris workstation I read the message. Then I open the first attachment with Acrobat Reader (a free program for displaying and printing PDF documents). There is an error on page 4 and it is neither possible to view it nor to print it. The 18 other pages are intact and it is possible to view them and to print them.
2. Accustomed to this kind of problems, I try to open the attached file with another implementation of Acrobat Reader, on a Windows PC. The same problem occurs: page 4 cannot be displayed, nor printed with this second tool. So the file must really be corrupted, and the fault is probably not in Acrobat; I continue my investigation with the next suspect: Netscape.
3. My first try to avoid Netscape is a short but unsucessful attempt to extract the files from the email using munpack, a tool to unpack messages in MIME (Multipurpose Internet Mail Extensions) format on Solaris.
4. Finally I decide to try another implementation of Netscape that's running on the Windows PC. This requires to forward the message to myself and to get it on the PC directly from the server. This time the PDF file is extracted properly from the message and both implementations of Acrobat Reader are able to display and print page 4.

It is not easy to impute a responsibility for this accident. The software used to send the message may have encoded the file incorrectly, or the implementation of Netscape on Solaris may by faulty. It is also possible that the MIME protocol that specifies how attachments must be embedded in email messages contains an ambiguity and that both implementations are correct although their interpretation of the protocol differ slightly. Although the MIME protocol is not an extremely complex one (it's just saying how to encode and concatenate several files in a single message), implementing it is already a considerable effort: an open source library[1] for parsing, creating, and editing messages in MIME format contains nearly 31000 lines of C++ source code! This is more than enough to let a few non-trivial faults and misunderstandings occur, especially when two separate organizations must implement it independently.

What could we expect from mobile code in this situation? If the sender of the message and myself had been running mobile code execution environments on our respective computers, he would have been able to choose how to encode the two files in the message, and to send me the program capable of displaying the text and extracting the attachments. It is not exactly the same program that encodes and decodes the message, but since they both have the same origin, we can expect that the decoding function is compatible with the encoding one, or at least that faults can be detected and corrected much easier. Furthermore, the program could have asked me to read, print or store the attachments on disk, and return to the sender in order to confirm the operation.

Without making wild speculations, we are already mentioning functionalities that standard-based systems have failed to provide for years... Such examples convince us that even if there are still problems to solve in current execution environments, and no real killer applications for mobile code, the approach is worth trying.

3.4 An architecture for services

A definition of **architecture** that is generally accepted in the software field is

> *... policy and rules stating how to define reusable software components, the structure that interconnects them, and the rules by which they interact and integrate.*

(Mowbray and Ruh, 1997, p. 215)

[1]http://www.hunnysoft.com/mimepp/

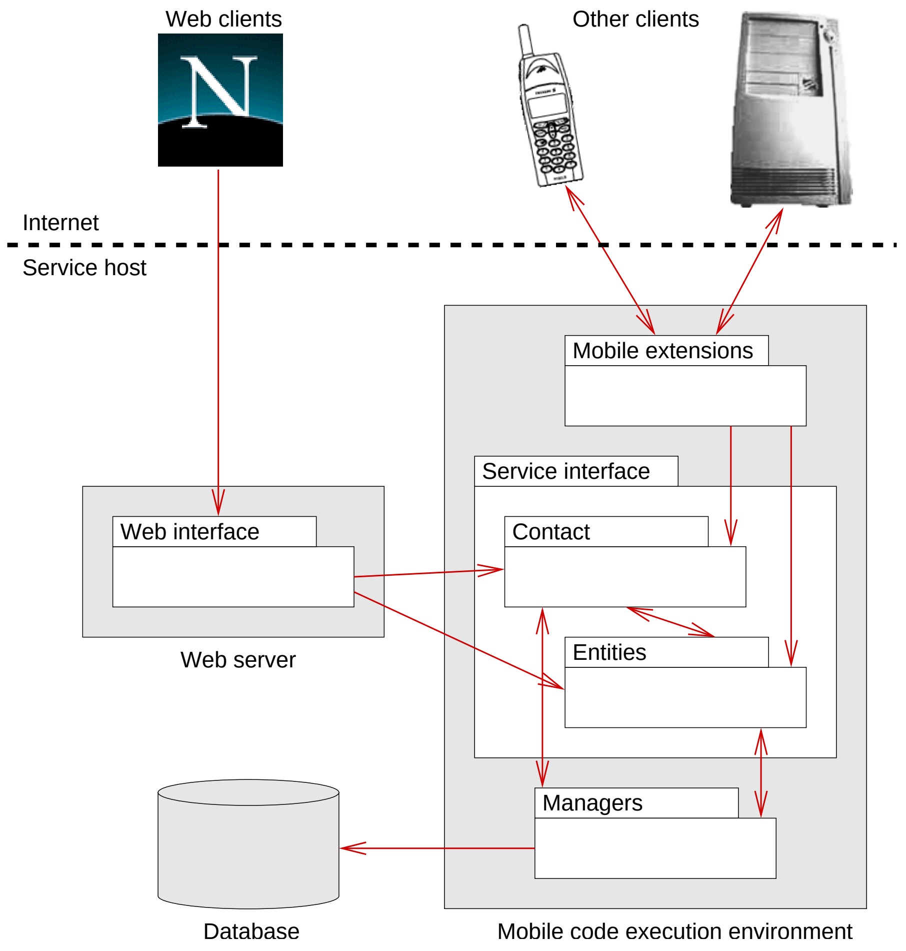


Figure 3.3: General architecture of a service within a mobile code execution environment. A "folder" represents a set of components (e.g. classes) of the same kind. Arrows represent the allowed dependences between components. For instance the database can be accessed only by "Manager" components which have the necessary information to make queries, and the database stands on its own, without relying on or "knowing about" other components.

Fig. 3.3, p. 60 shows with a high level of abstraction our architecture for a service that communicates using mobile code. The different parts are discussed in detail in the following paragraphs.

3.4.1 Database

A fundamental need of almost all information systems is a large and persistent storage for their data. They usually must store documents, information stored by users in the system, preferences or profiles of the users, as well as management information (authorizations, parameters, historic...).

Most databases at the time of this writing follow the relational model, where information is stored in tables, notably because of its good performances, and ease of administration. It is also possible to store data directly in files. For very simple data structures it could require less work, but it is often less convenient, simply because database management systems offer advanced facilities like atomicity or timestamps on entities, that would have to be programmed explicitly with standard file systems. Another approach is taken by object-oriented databases, where objects and graphs of objects can be stored directly in the database, without mapping their fields to tables. Using this model frees the developer from taking the aspect of **storage layout** into account and should also be possible with our architecture. Knowing which kind of database model is best suited to DEDIS goes beyond the scope of this dissertation.

3.4.2 Managers

The folder labeled "Managers" represents a set of software components (e.g. classes) that control access to the database.

Their primary responsibility is to carry conversions between the information that is stored in the database and the data structures of the execution environment. For instance, if the execution environment is a Java Virtual Machine (JVM) the managers convert rows in the database tables into instances of corresponding classes. These instances can then be handled by Java programs that implement the service or by mobile components.

The operations offered by managers are typically the three operations that alter the contents of the database: to create, delete and update entities, plus search operations that instantiate and return entities matching some criteria. The managers are also well suited to avoid unnecessary requests to the database using **caching** mechanisms: since they control how the database is accessed, they are able to know when the cache must be invalidated because of a change or deletion of information.

In addition to handle entities, the managers are also useful to handle the persistent relationships between entities, sometimes using additional tables. They are also well suited to enforce some integrity rules that can be programmed within the operations offered by the managers, but that would be difficult to handle at other places because they involve several entities or several tables.

At last, the managers may contain some active threads that periodically check the contents of the database, and generate events when incorrect or outdated information is found in the tables.

3.4.3 Entities

The entities represent the informations of the domain in a structured way (e.g. classes). Persistent entities are stored in the database, instantiated by the managers, and offered by the service to its clients. They may also be created by the clients and passed to the service which stores them in the database using the corresponding managers.

Unlike the managers, some entities may be accessed by programs outside the service, and as such, they belong to the service interface (hence the larger folder that contains the one labeled "Entities"). With them, clients must be able to retrieve, create, delete and update information in a structured way and to perform computations on this information (the functional aspects of the service).

The goal of entities is to store and present information in a simple, practical format, as close as possible to the data structures of the language (e.g. objects). This means for instance that presentation issues should be delegated to additional "interface" components, and not dealt with at the level of entities. This separation allows that several clients with different presentation needs and user interface models interact with the same entities. Separating data from presentation is also a much easier way to deal with the aspect of **internationalization**. It is also prevents costly conversions that inevitably occur if the entities return informations encoded in HTML or XML within text Strings. Moreover, relying on typed data structures gives the execution environment the possibility to perform useful semantic checks and to detect some programming errors. This is a standard recommendation in object-oriented software design (Jacobson et al., 1995, § 7.3.2).

Persistent relationships between entities should be handled by tables and managers, using "keys" like long integers, and not by associations (pointers) between objects. The obvious reason is distribution: a pointer is meaningless outside of a single address space. There are other reasons like **access control**: a client may access an entity but not those which are referenced, thus it is un-wise to have a reference within the entity object and methods that use this reference, even if they do not expose it directly to the client. It is much easier to perform access control if the client obtains a key and performs another request to reach the associated object. Other aspects that may be disrupted by references between objects are **caching** and **memory management**. Inconsitencies may appear if entities are inadvertently duplicated because the managers don't find the instance in their cache, or re-instantiate an entity that they had previously stored on disk to recover central memory. A useful rule of thumb is thus to avoid keeping reference to entity objects, unless the goal is precisely to create copies that can be changed temporarily, without affecting the database. A useful exception to this rule is when lists of entities are built by a manager. In this case it is too inefficient to query the database once for obtaining the keys and once for each entity, hence the list may store references to entities, but then, the whole list should be referenced only for a short period.

Entity objects are also useful to enforce some integrity rules that constrain the state of one entity, and ensure that the operations attempted by the clients are not violating these constraints.

3.4.4 Contact

This second folder inside the service's "software interface" contains the components that provide a contact point for clients and also the components that grant authorizations and that perform access control. Publishing a single contact point for clients, (e.g. an instance of a well-defined `Facade` class whose address can easily be found) as a means to interact with the service, is convenient both for the service provider and for the client. For the latter, this means that the service is easy to find and to access, and for the service provider, this is a way for example to hide a single instance of each manager, which can then be used in a controlled way to retrieve informations, update internal state, etc.

This level of indirection allows controlling access to the service in a centralized manner, and hides the managers, which is also a good thing because they can be changed if the database needs to be restructured, hopefully without changing the `Facade`.

There are many ways to perform access control, from simple schemes based on user names and passwords, to sophisticated mechanisms involving cryptography and certification authorities. It is more a matter of the level of security that needs to be achieved and of the support provided by the execution environment and can be chosen without fundamentally altering the architecture (which doesn't mean that individual components are not programmed differently, because of cross-cutting). In our case-study programmed in Java

(Part II) we have attempted to provide an acceptable level of privacy by having a method in the `Facade` that returns an `Authorization` when presented a username with the matching password. Other methods of the `Facade` that may return or modify a private information require the `Authorization`, can check which user is calling the method and reject the call if the user hasn't got sufficient privileges.

Information returned by the `Facade` is structured with the entities that were described in the previous section. For this reason we have enclosed both folders in the "Service interface", because entities returned by the `Facade` belong to this interface just like the `Facade` itself. This decision has implications from the point of view of security: the entities should not permit the client to perform operations that require privileges it doesn't have. To solve this problem, we decided that all operations requiring access control would be provided only by the `Facade`. It is certainly also possible to let entities perform this kind of security checks. In some cases, this required programming several versions of the entities, with different levels of detail and giving the detailed version only to the owner of the entity. Anyway, security is not the topic of this dissertation and many other ways to parameterize and enforce these security aspects are described in the literature, certainly with better protection and greater scalability.

Another essential responsibility of the components is to inform the clients when events occur inside the service. Thus it should offer some mechanism, that will typically based on the concept of "publish and subscribe", that provides clients the possibility to register when they are interested in an event, and to notify those clients when the event occurs. Otherwise, the clients necessarily have to query the service at regular intervals in order to know if something has changed. This active "polling" not only wastes computational resources but also introduces a lag between the occurrence of event and its detection that may cause many problems.

Because the event handler installed by the client doesn't belong to the service, some precautions are necessary to avoid that the execution threads of the service stay trapped inside faulty or malicious event handlers. In our case study, we used "one-way" method invocation, which is a way to execute the methods of an object asynchronously and concurrently. Another thread of execution is created to execute the method and the caller is not suspended but it can continue its own computations safely (see an example in § 9.3, p. 170).

Publishing many events requires a lot of work, but it is very important that the service publishes most internal events that may be interesting for the clients. Such events may be all creations, deletions, modifications of entities by the managers. The service may of course require that clients registering for an event notification possess sufficient access rights.

3.4.5 Web interface

In addition to its "software interface" (entities and contact point), the service needs a user interface (UI) that presents information in a human-readable form and lets the user accomplish his task in a convenient way. This second interface is managed by the components grouped in the "Web interface" folder. Using the standard Web protocols HTTP and HTML, these components are able to connect the client's Web browser to the service's software interface. When the client requests information, they transmit the request to the service, then they encode the result and send it back to the client. When they receive inputs from the user's keyboard and mouse they also interact with the service and return new "pages" that allow the user to continue its interaction.

To base the UI on Web protocols is very convenient because the interface becomes universal, in the sense that it can be displayed from any computer which is connected to the Internet and equipped with a standard Web browser. Thus the user is able to interact with the service from anywhere, without installing any additional software and without needing to configure the local system, since everything is handled by the service. This strategy is usually named **thin client** computing, because of the ability to use generic software that has a very limited responsibility of arranging visual elements on the display, while the computations that produce the contents and its logical layout are performed by the service.

Another advantage of Web-based UIs is that the pages shown to the user can contain a mix of data, images, text, and references to other relevant pages or documents. These other documents (help files, discussion forums, product comparisons) may even be located on other servers, and they may considerably enrich the value of information returned by the service. Conventional graphical user interfacess (GUIs) are usually much soberer with respect to textual, or contextual information. When such information is provided, it usually requires selecting a button or menu item, and another window pops up to display the relevant explanations. When everything occurs within the Web browser, the user can be conducted smoothly from one information to another, and even from one service to another.

The folder "Web interface" is not located in the execution environment but on a Web server for practical reasons: the latter is obviously much more capable to interact with the browsers than the mobile code platform. Nevertheless, components in this folder are strongly coupled to those of the service's software interface. In addition to presenting the information in a convenient way, they have the responsibility to control the sequence of interactions between the service and the user, and to manage sessions. Sessions typically start with user authentication and finish with some signing-off step. Inbetween, it is convenient to retain some state about the current session in Web interface components, to avoid transferring it with each PDU.

Web interface components are coupled to those in the service interface, hence the arrows linking their folder with Contact and Entities folders. However, the arrows are not bidirectional and components in Contact and Entities folders must be independent from UI components. The first reason is that Web technology changes extremely quickly, thus the service provider must be free to change the Web interface at any time, in order to adopt the technology that delivers the best functionality at the lowest cost. It is clear to me that I would have used Perl to write the UI components of a service created five years ago, afterwards I would have chosen Java Servlets, and now I would be very tempted to adopt XML-based tools, or something that integrates with the Web "portals" that may soon be adopted by institutions to deliver personalized content to their employees. During all this time, I could have kept the same service components untouched, thanks to the encapsulation provided by mobile code.

The second reason to have a service that is independent from its Web interface is even more important than the first one. Implementing the Web interface on top of the software interface is some kind of way to check that all the necessary operations for third parties that want to integrate the service with their own software are present.

One disadvantage of thin client strategies is that they are not well suited to display complex data. They also require many interactions between the browser and the Web server. Over slow connection lines, this may result in poor responsiveness. It is however possible to reduce the number of interactions or to render richer data representations by moving part of the computations from the server to the client host. Commonly used solutions are scripts which have a low overhead and the advantage that they do not disrupt the users' habit to browse "pages". Another solution is Java Applets, which have a slightly higher overhead, and are not so well melted into pages. The highest performance may be achieved by having a platform-dependent client, compiled for the user's operating system, but this has the highest installation overhead. The big drawback in all these strategies is that they are sensitive to differences in the execution context and are much less universal than HTML. However, the architecture guarantees that they can all be tried without having to change the core of the service.

3.4.6 Mobile extensions

The folder "Mobile extensions" represents all foreign software entities (programs with their own state and threads) that may come an go in the execution environment and run close to the service. They interact with the service through its software interface, using exactly the same components and functions as the Web interface.

Instead of choosing a high-level communication standard like Java RMI (§ 4.5, p. 81) or CORBA (§ 4.6, p. 85), the service provider assumes compatibility only at the most common level, for instance TCP/IP. It also assumes that third-parties are able to write and send programs in the language that the execution environment is able to interpret (e.g. Java bytecodes). For all higher-level needs, he lets clients choose their own interaction protocols and encapsulate them in mobile code. Like the aforementioned standards, this provides openness, but without committing to one standard, which may not remain a standard eternally, may contain ambiguities or may be implemented in several incompatible ways (see the MIME example in § 3.3, p. 56). Furthermore, it doesn't impose a standard to clients which may not want or may not be able to follow it.

Obviously, it is not possible to avoid that several kinds of incompatible execution environments become used. It is not a big problem with our scheme since, unlike other authors, we don't expect that mobile components visit many hosts during their lifetime (§ 4.4, p. 79). What matters here, is that as long the service is available on the same platform, interacting with it will remain possible, even if there are big changes in communication technologies, or in the non-functional aspects that indirectly reflect the features of a completely new environment. The only requirement is that it remains possible to send a new program to the platform, which is suited to the new environment. For instance this would require that a suitable version of the Java compiler and a TCP connection remain available.

For the developpers of third-party services, being able to install and execute programs at any time, on the service's platform, offers the possibility of interacting with efficient local mechanisms, which are not subject to the problems of physical distribution.

Extensions are also especially useful for users that do not maintain a permanent physical connection with the service, for instance because their computer is turned off most of the time. An extension is a means to achieve a permanent virtual presence, in order to respond to events with exactly the behavior required by the user, without any collaboration with the service provider.

Depending on the facilities offered by the execution environment, supporting extensions may require some specific mechanims in the service's software interface (Contact folder), for instance to preserve extensions when the execution needs to be restarted. In Chap. 9 we present concrete extensions and such useful mechanisms, that let extensions inform the service that they are present, without revealing their address to all other extensions, which may not be trusted.

This architecture is strongly open: users that are not satisfied by the Web interface, as well as other services that require better control, have the permission to run their extensions on the service host, inside the execution environment. Given the quick changes in technology and needs, this is a real advantage for the service provider since little collaboration is required. It is also interesting for people used to the system, that have already stored information in it, because it reduces the risk that they have to switch to another service because of limited behavior and functionality, or diverging technologies or protocols.

3.5 Features of an execution environment

Many mobile agent platforms have been developed in the recent years (58 available entries in the Mobile Agent List[2] in December 2000). Although they all share the common goal of running programs that move from computer to computer, there are such great variations in the features and mechanisms offered that it is very difficult to distinguish the essential ones from those which could be useful, and from the superfluous ones. One such hotly debated feature is the so called "strong mobility": the ability to move a running thread from one host to another, with its full state, including the program counter and stack of "frames" (parameters and local variables of currently invoked methods). Some authors think it is a valuable feature and are working hard to incorporate it in their software, others argue that there are only a few points where threads are actually ready to migrate and that it is sufficient to restart them from there, and to let them choose explicitly which part of the state must be preserved.

To present the main features of an execution environment, without getting lost in details, we base this section on the `MØ` Messenger platform and language (Tschudin, 1994; Tschudin, 1997a). For our purposes, it has the advantage that **minimality** was constantly sought in its design. Thus, it implements the Messenger paradigm, and supports code mobility with a set of features that is reduced to the essential. The second force that has strongly influenced the design of `MØ` is **scalability**, thus all the available mechanisms should still function when millions of platforms are interconnected. Features that could be useful when only a few platforms are involved but that cannot be managed at Internet scale are carefully avoided. For this reason, all facilities offered by the platform must be local and work without any kind of centralized control, and everything that involves more than one platform must be built using mobile code.

[2]http://mole.informatik.uni-stuttgart.de/mal/mal.html

3.5.1 Messengers are executed as anonymous threads

When a `MØ` platform is running on a given host, it listens for incoming messengers on a predefined set of channels (e.g. UDP ports). Well-formed messengers that arrive on these ports are executed in concurrent threads. These threads are anonymous, there are no "handles" or thread identifiers that could be used to address or act on a thread. Anonymous threads are also more difficult to attack, and thus they are justified from the point of view of security. The problem of assigning unique names is also avoided.

A well-formed messenger is made of a key that identifies the **queue** in which the resulting thread must be inserted (see below), of the **code** of the messenger (a string of `MØ` instructions) and of an optional **data** field. The `MØ` language contains an operator to create well-formed messengers, and an operator to submit new threads, either on the local platform, or on another platform that can be reached across the network. There are other operators that a thread can use to determine where it comes from, or to manipulate the code field, the data field, and the whole messenger as strings.

3.5.2 Shared memory enables local interactions

Threads that need to interact can exchange informations using the platform's global dictionary, a shared-memory area that can be used in a standard way on every `MØ` platform. There is no way to exchange messages between two threads that are not co-located. Thus, if an information must be communicated across the network, it must be serialized in the data field of a messenger that travels to the remote location, deserializes it, and makes it available in the global dictionary of the remote platform.

Dictionaries are data structures that map keys to values. The operations associated with dictionaries are: (1) inserting a new ⟨key, value⟩ pair into a dictionary, (2) retrieving the value associated with a key, (3) removing a ⟨key, value⟩ pair, (4) checking whether a key is present, (5) retrieving all pairs in the dictionary.

Apart from dictionaries, `MØ` offers several data types, such as integers, strings, arrays, as well as names (arbitrary sequences of characters) and secret keys (sequences of 8 bytes). Operators can check the types of their operands. However, there are no pointers in the language, to avoid that threads access memory that is not within their local memory area, or that has not been explicitly shared through the global dictionary. There is also no notion of class or object in the language.

3.5.3 Queues enable coordination

To coordinate their activities, threads can use the powerful mechanism of queues offered by the platform. The principle is that at any time, a thread can be in at most one queue, and that a queue can be either in the stopped state or not. Threads that are not in a queue are always ready to run. When several threads are in the same queue, only the front-most is authorized to run. When this thread leaves the queue or terminates, the state of the queue determines whether the next one may proceed or not. If the queue is in the stopped state, the thread stays blocked, otherwise it is authorized to proceed, and even if the queue is subsequently stopped, the thread will continue to execute.

With the provided operations, a thread can (1) enter a queue, identified by a key, (2) leave the current queue and execute outside of any queue or enter another queue, (3) change the state of a queue, (4) obtain the key that identifies their current queue. When a thread enters a queue, it could optionally specify a timeout value. If the thread has not been permitted to run when the timeout expires, the thread is automatically removed from the queue, receives a notification and may continue.

Queues are a simple yet powerful mechanism to synchronize threads, they are provably equivalent to other well-known coordination mechanisms like semaphores (Tschudin, 1993).

3.5.4 Local mechanisms enable security

Instead of trying to authenticate the source of an incoming messenger, and rejecting untrusted ones, `MØ` platforms unconditionally execute all incoming well-formed messengers. The reason is that authentication schemes require protocols and agreements between the people who operate the platforms, those who provide services and those who send the messengers and hence are out of the scope of the messengers. Like for other problems that cannot be handled in a local and scalable way by the platforms, the approach of `MØ` is to provide low-level mechanisms only and to let messengers protect themselves and implement the different security policies that are required at the application level.

These mechanisms include: (1) using random keys that are very hard to guess (8 bytes) to identify queues and dictionary entries, which are also a means to avoid name clashes, (2) fine grained access rights for every reference on global data: read, write, and execute attributes with operations to remove and check them, (3) no possibility to browse the global dictionary, thus two threads can exchange data using a secret key and no other thread can find it, (4) operators for DES cryptography and MD5 secure hash, (5) use of assymetric cryptography such that only one platform can execute a given messenger:

the messenger is encrypted with the platform's public key and submitted in the encrypted form, such that the private key is necessary to decode it, (6) secret define operation to add a pair to an unprotected dictionary in such a way that it can be removed only by threads sharing a related secret key.

3.5.5 Market-based resource control

The mechanisms above allow the implementation of some form of access control, but they do not protect the platform or services from a whole category of security attacks: the so-called "denial of service" attacks. Such attacks are based on the fact that even if a service protects its data and functionality from illicit access, it can easily be made unusable by overloading it with a huge number of irrelevant requests. When all of its computing or network resources are consumed by fake requests, the service is no longer able to carry its normal task. Such attacks are reported from time to time, causing a lot of flurry when they are aimed at big commercial Web sites such as Amazon or Yahoo, and potentially cause huge financial losses.

It is easy to see that executing untrusted foreign code represents a increased potential for such attacks. A malicious extension may for instance consume all available memory or spawn so many threads that all other activities are intolerably slowed down. Hence, platforms need reliable mechanisms for resource accounting and control.

In M∅, the consumption of all resources is billed, using a fictitious currency, which has a value only inside one platform that is also able to create it (Tschudin, 1997b). When a messenger enters the platform, it receives an initial credit, available on its default account. The computations performed by the thread or its descendants, the memory they use, and the packets they send on the network all decrease the amount available on the account. Once the account is empty, the platform kills the threads that depend on it and recovers the memory. Each platform adapts the prices of resources as if they were goods on a market: as long as the demand is low, the prices are low, and when the demand increases, the prices get higher and the processes must adapt, either by reducing their consumption or by moving to cheaper platforms. The goals are of course to achieve a good load balancing among available platforms, but also that messengers providing useful services, which they can offer to others in exchange for some currency, have an additional source of revenue and can afford more resources than those which are not providing any useful services.

There are operators in M∅ to create an empty account, to draw a cheque from a non-empty account, to deposit a cheque on an account, to find or set a thread's default account, and to add or remove a sponsoring account to an item in the shared memory (since it is shared, it is potentially interesting for several threads, and if one of them disappears, others can continue sponsoring the item).

Another possibility would be to link the fictious currency to actual money, and thus to integrate the non-functional aspect of payment. Unlike all the previous mechanisms that have actually been implemented, this is just a scenario. The platform could for instance periodically allocate a certain amount of its fictious currency on a predefined account against payment of a small amount of real money. Thus, only messengers that are actually sponsored by their owners would be able to use the platform's computational resources. In this context, denial of service attacks become much less likely and somehow more acceptable, since they cost real money to their authors, and represent a tangible income for the "victim". The new forms of electronic payment with very low transaction costs and the ability to handle very small amounts of money (e.g. PayPal[3]) make such payment schemes feasible.

With or without bindings to real money, this market-based mechanism is definitely a powerful one to recover unused resources: as soon as a distributed application stops feeding its components (threads, memory), the platforms know that they can delete them, and reallocate the corresponding resources to other processes. From a practical point of view, this represents an interesting alternative to the distributed garbage collection algorithms found in other systems.

3.5.6 Other useful operators

M∅ provides a few other useful facilities that should be available in all execution environments. These operators are a means to retrieve the current system **time** (UTC to avoid problems with platforms located in different time zones) and to perform some arithmetic operations on this particular type. Messengers may also obtain the id of the current **host** and of the current **platform** (there can be several platforms running on the same host). Last there are minimal facilities for **logging** and **error handling** (threads that contain programming errors and do not handle them get killed without notice). All other elements in M∅ are traditional imperative programming constructs like loops, procedures, etc.

Chapter summary

In this chapter, we described our approach that exploits mobile code for the design of extensible services. In the next chapter we will compare our proposition with existing solutions to show (1) that our architecture contributes to

[3]http://www.x.com/

a better understanding of how large-scale distributed applications should be designed, notably at the level of services, and (2) that using mobile code in this context is fully justified, given the large number of aspects that it allows to handle, with a minimal number of conventions. The case study of Part II will also give further support to these claims.

Chapter 4

Related work & state of the art

Chapter highlights

- The alternate architectures: actual systems and services that have been built around architectural principles similar to those of the previous chapter (StormCast, Tabican and eAuctionHouse).

- Why there are no mobile agents roaming the Web: still too many problems with trust and ontologies.

- The alternate integration technologies: technologies that share similar goals of making the development and integration of services in a distributed and dynamic context easier, but that don't give the same importance to mobile code (Java, CORBA, XML).

Note: In order to keep the length of this chapter whithin reasonable bounds, we are not going to describe or compare the many existing mobile code execution environments because several very good surveys on the topic have been published recently and are easy to obtain on the Web (Papaioannou, 2000; Lugmayr, 1999).

4.1 StormCast

StormCast[1] is an existing system proposed by the TACOMA research team of Tromsø and Cornell Universities (Johansen, 1998). The goal of StormCast is to provide weather information and forecasts. What differentiates StormCast from other weather information sources is that the service provider has programmed a set of agents for their customers who can choose parameters and activate agents according to their own needs. These agents are equivalent to our extensions. They can interact locally with the main service, even when the customer is not connected to the network. The parameters of the agents allow that each customer chooses which weather situation is relevant for him: a yachtman could decide that a well established wind and no rain is the perfect weather for a day off, while fishermen and windsurfers each will have thier own sensibilities to wind and rain conditions. The customer can also choose by which media the system is going to inform him when a relevant situation is likely to occur: email, pager, fax... Once the user has activated an agent, the server periodically runs it to check the actual information and to send a message to the customer if necessary.

According to (Johansen, 1998) the need to build extensible servers for StormCast was a strong motivation to investigate mobile code and to develop TACOMA. The same paper reports some interesting performance estimations: it takes approximately 1.5 second to install an agent from a Web browser, through CGI scripts, and 1000 agents performing a simple computation (checking a temperature and comparing it with a threshold value) can be executed in about 8 seconds on today's desktop PC's (200MHz Pentium Pro with 128MB RAM).

The architecture of StormCast is described in a recent technical report (Johansen et al., 2000). The essential difference between their architecture called ACE (Agent Computing Environment) and ours is that TACOMA supports program mobility but doesn't require that agents run inside an execution environment. The agents may run directly in standard operating systems (e.g. Unix) and they can be written in any language supported by a compiler or interpreter for this operating system. This language independence can be exploited for writing different parts of the system with the most appropriate language. On the other hand, allowing that unrestricted programs access all the functions of the operating system implies that these programs are trusted: they don't try to steal information, and they are willing to share system resources. In our architecture, the execution environment is able to enforce resource control at a low level, but no such mechanism is available in most operating systems. Thus, applications built according to their ACE should

[1]http://www.cs.uit.no/forskning/DOS/StormCast/

contain a layer for accounting (L3—Extension Layer) and another one for load balancing and scheduling (L5—Mediator Layer).

Another difference between the ACE architecture and ours are the higher-level layers that provide the possibility of assembling components in order to create agents, to customers who are not trained programmers (L7—Compose layer); and a layer that enables the deployment and control of agents through a GUI, by the customer (L8—User layer). On the other hand, contrary to ours, their architecture doesn't describe how the servers can be designed. It just suggests that there is a Server Layer (L1) where standard message passing is used to interact with an open application.

An older technical report (Hartvigsen et al., 1994) describes an architecture for StormCast that is much more similar to ours, especially with its separation of "data storage", "information" and "user interface" layers which roughly correspond to our database, entities and Web interface folders. On the other hand, we see that in this precursor of the actual StormCast service, the potential of mobile agents had not yet been discovered, even if the necessity to extend the service had been integrated in the architecture by means of a "utility" layer where several "tools" could be started, stopped and queried.

Interestingly, there are two additional layers at the bottom of their architecture that result naturally from their weather forecast application: they need software for monitoring information sources and collecting relevant information that gets fed into the data storage layer. In our architecture, we have not particularly studied this kind of function, but we can imagine information that is collected and buffered on another host and enters the service through its software interface. This ensures that monitoring and collecting functions are not influenced by the load of the service's host, and thus to cope with the non-functional aspect of real-time constraints. This also keeps the "managers" simple and avoids that small changes in the information sources require significant changes in the service internals.

4.2 TabiCan and e-Marketplace

TabiCan[2] is a real commercial service in the domain of travel, developed by researchers of IBM Tokyo laboratory (Yamamoto and Nakamura, 1999). It relies on Aglets[3], a Java-based mobile agent platform. The TabiCan server allows the interaction of the independently-written agents of customers and shops.

[2]http://www.tabican.ne.jp/, japanese only!

[3]IBM has released the Java source code of Aglets which is available either on http://www.aglets.org/ or on http://sourceforge.net/projects/aglets/. The system is described in (Lange and Oshima, 1998).

Customers, which use their Web browser to find airline tickets and package tours, can obtain information from the shops of several travel agents in a single search action. Just like in the case of StormCast, the customer's action allows the instantiation and parameterization of a Customer agent that subsequently works on the TabiCan server. Since searching the best deal from many shops potentially takes a long time, search in TabiCan is asynchronous. Customer agents live during two days and can be accessed several times, as long as they are alive, even if the user has switched his computer off inbetween. The user can either browse the results after some time, or be notified by email when the results are available.

The authors of TabiCan do not expect that all travel shops can or want to install their Shop agents on a single server. Thus, their system accommodates several similar servers, that can be visited by mobile agents during the search. The Customer agents themselves are not mobile because they must be easy to reach and respond when their owners want to know the status of their requests. But they can send "slaves" to different servers and thus communicate their owner's requests to all affiliated shops. Having several servers with a limited number of Shop agents on each one is a good way to balance the overall load, because there are potentially thousands of Customer agents and hundreds of Shop agents that want to interact.

In (Yamamoto and Nakamura, 1999)—the only description of TabiCan that we were able to obtain—neither the Customer agents, nor the Store agents are described in great detail. But the paper contains some interesting descriptions of the environment in which these agents execute, and how they communicate. Like TACOMA, the Aglets system does not take the aspects of accounting and resource control into account. Thus, some components in the architecture are responsible to ensure that agents get fair opportunities to execute and that the server hosting them is not overloaded. This would easily be the case if 2000 Consumer agents requiring 30 kbyte for their code and state, and one or several execution threads each, would be active at the same time.

This solution is implemented in a library called e-Marketplace, which is built on top of Aglets and controls the use of memory and CPU. It supposes that agents interact only by message passing, through the library, and according to a protocol chosen by TabiCan developers. The types and order of messages exchanged between Consumer and Shop agents are fixed. This enables that the two main components of e-Marketplace control which agents must be active (others can be swapped to secondary storage), and also schedule their execution. Processing one request from a message queue is the basic "unit" of execution for an agent. The library is also able to manage a fixed-size "pool" of execution threads, which iteratively select an agent and process one message.

This study clearly demonstrates that efficient mechanisms to control server resources are indispensable when large numbers of agents are involved. However, implementing these mechanisms at the application level, or with an additional library as in TabiCan, strongly limits the computations that agents may perform, or the ways in which agents interact. Furthermore, the mechanism could be unable to effectively prevent excessive resource consumption, when foreign, untrusted agents are involved. Hence our choice to rely on resource control mechanisms within the platform, and not at service level.

4.3 eAuctionHouse and Nomad

eAuctionHouse[4] is an auction server where users across the Internet can buy and sell goods, and set up markets (Huai and Sandholm, 2000). Nomad is the mobile agent system integrated with eAuctionHouse, and is itself based on Concordia[5]. Like Aglets, Concordia is a Java-based mobile agent platform. eAuctionHouse is a free-to-use third party auction site which provides a wide range of customizable auction types.

Using their Web browsers, users can buy and sell items, and set up markets. eAuctionHouse implements some features which are not found in any other electronic auction site on the Internet, like bidding on combinations of items or bidding with price-quantity graphs. Users can create or close auctions, submit bids or manage agents that participate in auctions on their behalf. These agents are able to collect information, to send this information by email, to learn price distributions, to bid, or even to set up auctions. They can represent the user inside the eAuctionHouse, even when he is not connected to the network, and free him from actively watching his auctions. They can also implement his personal bidding strategies and since they are permanently on the server, they can act on his behalf without the delays that could result in missed opportunities.

Agents can be written directly in Java, or chosen from Nomad's template library. Web forms offer the choice among 5 existing templates, then are used to parameterize and start the agent. Templates implement well known strategies and help novice bidders compete with experts.

The architecture of eAuctionHouse is similar to ours, it combines an "Auction database", an "Auction engine" that corresponds to our service entities, as well as a Web interface. There are a couple of additional components ("Agent generator", "Agent manager") for the management of agent templates, and

[4]http://ecommerce.cs.wustl.edu/eAuctionHouse/

[5]A free evaluation kit, without source code, is distributed by Mitsubishi at http://www.meitca.com/HSL/Projects/Concordia/, a commercial version with additional reliability, security and administration components is also available.

that let the user interact with his agents through the Web. Our architecture doesn't offer such facilities to communicate with the extensions, nor a repository for agent templates. Although these functionalities are important, we don't think that they need to be implemented by each service and we rather see them as useful add-ons that could be made available by third parties and reused by several services.

Another similarity with our architecture is the "Agent dock" that must be used by agents to inform the service that they are present. We have also observed that the service needs to know which extensions are currently working on the local platform, especially to avoid losing them if the platform must be stopped and restarted (§ 9.3, p. 170). However, our architecture doesn't require an "Agent database" where agents are persistently stored with related information (owners, event notifications that they want to receive, etc.) because we exploit mobility to save agents during shutdowns.

The authors of eAuctionHouse also envision that not only one, but many eAuctionHouse servers will be deployed, and that agents will be able to relocate themselves in order to balance the processing load, and minimize network traffic and delays. However, moving will not be compulsory because the eAuctionHouse servers also accept bids and transactions as messages formatted according to a predefined protocol and received on a TCP/IP port.

To interpret such messages, a "Connection manager" component is inserted between the Auction engine and other entities like the Web interface, the agents, or the external processes communicating through the TCP/IP port. Without a better description of this Connection manager and of the protocol that defines its behavior, it is hard to guess how complex it is and what it does exactly. Anyway, our architecture discourages the service provider to define such a protocol to interact with the service because it is practically impossible to define a protocol that satisfies all clients. It also avoids that such a static protocol component is embedded inside the service because a huge effort is needed to implement it (Queloz and Villazón, 1999, §4.3) and because of the difficulty to change it if necessary. We believe that it's much more fruitful to define a good and stable Service interface that can be accessed at a "procedural" level, and to let each client encapsulate his favorite protocol with a mobile extension, even if the extensions are developed by the service provider himself for practical reasons.

4.4 Roaming agent illusion

An idea that has been greatly oversold in the last years is that mobile agents would be able to roam the Web and look for information on behalf of an end-user. Such agents were supposed to perform filtering tasks, or to visit several sites, looking for the best offers for a good that their owner wants to purchase. They should have transported credit card numbers and performed the whole transactions while the user could be able to relax or perform more important tasks. Someone else would have programmed the agent, and it would have been able to learn about user preferences thanks to close interaction during a long period of time. Hence, the users would save a lot of time otherwise spent surfing to find the best offers.

Although some interesting solutions have been proposed to enable agents visiting Web sites in a secure way (Fünfrocken, 1998), we have seen no evidence that such solutions have been implemented and actually deployed. There are of course many "bots"[6] performing all kinds of useful work: indexing Web pages, comparing prices of online shops, etc. but they are not mobile. They access information located on remote sites using the usual HTTP protocol and they process it at their "home" site. This implies moving lots of data, and although convincing studies have shown that using mobility in this context is a means to save bandwidth and improve overall processing time, it cannot be done, simply because there are no mobile agent platforms on the Web servers!

The reasons why there are no mobile code platforms on Web servers are: (1) people writing the software for Web servers and people operating them have not yet perceived that the main benefits of mobile code are better integration and extensibility; (2) the problems of security and resource control are not yet solved in a satisfactory way. For the administrator of a busy Web site, that may already have trouble ensuring short response times, mobile agents must resemble a "bunch of viruses" and it is certainly unthinkable to let them consume precious resources for their mysterious computations. Actually, none of the systems that we have reviewed in the preceding sections—and ours makes no exception—tells how to run foreign agents on their sites, even if the possibility exists and the owners of the sites do use mobile agents for their own experiences. The reason is that with current mobile code platforms, resource control is insufficient, hence only trusted foreign code is acceptable. Better accounting and resource control mechanisms are definitely required (§ 3.5.5, p. 71).

Another problem that will probably hinder "roaming shopping agents" for an even longer time is the lack of ontologies. It is probably neither feasible nor desirable to standardize at Web scale the description of all products and of their attributes, selling prices, etc. Shopping bots may be able to uniformize the informations from tens of shops in order to allow product comparison, but this requires a lot of dedicated and hand-crafted software. This software must be updated frequently and is probably too large to be transported with the agent. Last, there are probably details in the attributes of some products, or

[6]http://www.botspot.com/

in the sale conditions of some merchants that the agents would not be able to grasp and that will always require careful examination by the customer.

4.5 Distributed objects in Java

With its Java language, virtual machine and extremely rich set of standard libraries, Sun Microsystems, Inc., has created a technology that nearly fulfills, and sometimes exceeds, all the requirements we have expressed for an execution environment. More remarkably, the technology has been adopted by a very large number of developers and organizations across the world, far more than any other platform, either academic or industrial, that would have offered similar features. However, it is not the ability to extend services with mobile code that has prompted this wide adoption but more simple features of Java like:

- the fact that Java programs are easily portable across different operating systems, thanks to the wide availability of standard virtual machines and a rich set of standard libraries;
- its simplicity, due to its object-oriented nature and the fact that error-prone features like pointers have been avoided;
- dynamic class-loading to incorporate the code of new classes into a running application whenever they are needed, possibly from a remote location;
- sandboxing mechanisms that attempt to ensure that untrusted code has access only to benign operations;
- etc.

Furthermore, Sun has been able to understand the importance of networking, and to incorporate several useful features for distributed programming directly in the language (e.g. threads to deal with asynchronous events) and in the standard libraries. Thus, with Java, the designer of a service is free to choose among several communication models and techniques:

1. **PDU-based communication**

   Classes in the "java.net" standard library offer a good assistance for the exchange of messages on top of IP protocols. Low level protocols such as UDP (connectionless) and TCP (connection-oriented) are well supported, making it easy for an application to listen for messages coming on sockets or to send back answers. The library also contain classes able to communicate with a Web server using URLs and the HTTP protocol, and thus making it easy to read a Web "page" inside a Java program.

Of course, it is up to each communicating party to interpret the meaning of the messages exchanged according to a predefined protocol. As we already mentioned several times, the drawback of this apporach is the large amount of work required to define protocols, write the message interpreters, and coordinate the updates that are necessary to remove faults, or incorporate new features.

2. **Remote Method Invocation**

   When Java objects are present on both ends of the communication channel, the designer can choose to let them interact using Java RMI mechanism. The principle is that instances of classes following a certain number of simple conventions can be exported by a server written in Java. This requires running a standard "rmiregistry" daemon on the same host. Exported instances are bound to a predefined name within the registry. Afterwards, clients running on remote JVMs can lookup registered instances by providing the predefined name, and invoke their methods remotely. The proxy obtained by the instance is called a "stub", and there is a corresponding "skeleton" on the server side. Stubs and skeletons are generated from the source code of the remote-enabled class using a standard "rmic" tool. When the client invokes a method on a stub, the arguments are automatically serialized, and the call is forwarded to the server-side skeleton, which in turn deserializes parameters and forwards the call to the actual remote instance. The client's thread is suspended until the call on the remote instance terminates and a potential return value has traveled along the opposite path.

   Thus RMI offers a procedural mechanism, that hides the underlying network and is not too complicated to use. It spares the implementation of message interpreters, which can quiclky become complicated, for instance when a client must interact with several objects of the service at the same time. It also relieves the programmers to encode parameters and return values since there is a built-in serialization mechanism that can cope with complex graphs of interdependent objects. Last, some simple coordination is provided by the system: since remote calls are blocking it is not necessary to suspend the threads waiting for an anwser.

   Java's remote classloading mechanism is used at two places in the RMI scenario: (1) the stub that the client obtains is dynamically loaded and can be obtained from a location that is under control of the service provider, thus different services that implement the same interface may well have different code for their proxy; (2) the parameters and return values of the methods, which must be serializable and are passed by value, may also be instances of classes that are not available on the remote platform, in this case the code of these classes can be downloaded from a remote location. For application programmers, the second case is interesting because clients can pass to the server a specific implementation of a predefined generic behavior which is specified by a common interface. This simple form of code mobility provides a way for the client to instruct the server how to perform some special tasks, or reciprocally.

Note that this ability is not specific to RMI and that the remote class-loading can also be programmed explicitly. In (Queloz and Villazón, 1999) we describe how we exploited this feature to program a generic server that can be instructed by its clients how they want it to encode the results of their requests.

3. **Runnable mobile objects**

   The mobile objects described so far are passive, they are passed around in order to satisfy the communication needs of clients and servers. However, they are already very close to the messengers that have the ability to start their own threads on available platforms for their own purposes. Since the state and the code of mobile objects is moved from one platform to the other, it is sufficient to spawn a new thread from one of the mobile object's methods to achieve the same effect.

   This is the reason why so many mobile "agent" systems are based on Java: with the available mechanisms, only a minimal set of additional conventions and programs are necessary to implement the abstraction of active objects, autonomously moving between platforms to achieve their tasks. The main benefit in using these systems is that they allow to work with abstractions of a higher level, succeeding to hide the details of the underlying Java mechanisms with various degrees, and that they provide additional facilities like asynchronous messaging, group communications, or agent tracking. Voyager, the platform that we have used for our case study belongs to this category.

4. **Jini infrastructure**

   Another part of Java that has been much publicized is Jini (Sun Microsystems Inc., 1999). The problems solved by Jini occur in very dynamic and heterogeneous environments where devices that communicate with different protocols and that have never been parameterized to work together must be able to cooperate. This is typically the case with mobile devices that are carried around and that must interact with whatever other hardware is available in the surroundings, e.g. a visitor's laptop that must send a document to a locally available printer. In the RMI scenario, the client is able to download the stub that can be used to invoke one of the service's method in order to print the document. Unfortunately, this supposes that the client knows (1) the network address of the RMI registry, (2) the name that has been used to export the instance that performs the print operation. The purpose of Jini is precisely to solve these two problems without having to parameterize the client with this information.

Jini specifies:

- how services and clients can find a **lookup service** without knowing anything about the network by broadcasting a **discovery** message to a standard port;
- how services can register with the lookup service, in order to describe their attributes, and to provide a **service object**, which knows how to interact with the service (playing the same role as the "Interface" instance in Fig. 3.2, p. 57);
- how clients can retrieve service objects matching their needs by querying the lookup service.

Thus, the Jini framework uses protocols to connect clients to servers and Java mobile code facilities in order to allow that services specify how clients shall interact with them. Service providers are able to encapsulate proprietary protocols between the service object and the corresponding device. Communications take place in the client's address space through calls to the methods of a well known procedural interface. This is certainly a legitimate use of mobile code, since it is a way for competing vendors of devices to keep their trade secrets, while providing a uniform and parameterless access to devices of various provenance.

From the above description of Java's strenghts, it could seem that it is the perfect platform for building distributed applications. However, there is one big shortcoming in the current implementations that still prevents a wider adoption of mobile code: Java has no mechanism for accounting and control of resources. There are elaborate security facilities ranging from authentication and access control (Lai et al., 1999), to the management of security policies that limit the operations performed by untrusted or partially-trusted code, but they are not sufficient. For instance an untrusted Applet may not access the file system; but nothing prevents it from creating thousands of threads, or from requesting megabytes of memory, thus blocking all other activities in the virtual machine.

RMI and Jini have also adopted a “leasing” mechanism, that grants objects for a definite period of time after which they are cleared if they have not been “renewed”. It has some similarities with the market-based mechanism of § 3.5.5, p. 71, but it doesn't prevent denial of service attacks. Better accounting and resource control are definitely needed, and are currently the subject of much ongoing research (Suri et al., 2000; Binder et al., 2000), as is the whole topic of market-based resource control.

4.6 Distributed objects in CORBA

A serious alternative to Java for the construction of distributed systems is CORBA (Mowbray and Ruh, 1997). It is a large set of standards and specifications resulting from the efforts of many players in the software industry (the Object Management Group, or OMG). The two main goals of CORBA are: (1) inter-operability between programs running on different operating systems and written in different languages, (2) reducing the costs of developpment and integration in large scale distributed systems by enabling a marketplace for reusable components that handle functional aspects in specific domains (e.g. finance, healthcare, transportation) as well as non-functional aspects (naming, security, concurrency, etc.).

An impressive list of Formal (Finalized) Specifications is available on OMG's Web site[7] (December 2000):

- CORBA/IIOP 2.4 Specification
  defines the object model, the structure of an Object Request Broker (ORB), the structure of a client, the syntax and semantics of the Interface Definition Language (IDL), the interactions with an ORB, the operations on object references, context objects[8], policy objects, standard exceptions, interface repositories, the General Inter-ORB Protocol, interceptors[9], asynchronous messaging, Real-Time ORB, etc.
- OMG Modeling Specifications
  1. Unified Modeling Language (UML)
  2. Meta-Object Facility (MOF)
  3. XML Metadata Interchange (XMI)
- CORBA Language Mapping Specifications
  1. Ada Language Mapping

[7]http://www.omg.org/technology/documents/formal/

[8]Context objects represent information about the client, environment, or circumstances of a request that are inconvenient to pass as parameters.

[9]Interceptors can be interposed on the invocation (and response) path(s) between a client and a target object.

2. C Language Mapping
3. C++ Language Mapping
4. COBOL Language Mapping
5. IDL to Java Language Mapping
6. Java to IDL Language Mapping
7. Lisp Language Mapping
8. Smalltalk Language Mapping

- CORBA Services Specifications

Name	**Goals**
Collection Service	*provides a standard library to handle collections of objects*
Concurrency Service	*provides locks to handle the coordination aspect*
Event Service	*provides multicast and a general mechanism to bind event sources to event listeners*
Externalization Service	*provides support for the serialization aspect, but cannot provide the nice integration with code mobility present in Java RMI because of language and platform independence*
Inter-operable Naming Service	*provides "yellow pages" to locate services*
Licensing Service	*attemps to enable the aspect of payment and enforce access policies in other people's enterprises, without a direct presence*
Life Cycle Service	*defines how factories could be used in order to take distribution, transparency, and language incompatibilities into account, and to allow the creation, moving, and destruction of instaces in an application-controlled way*
Notification Service	*extends the Event Service*
Persistent State Service	*provides object persistence*
Property Service	*provides support related to the parameterization aspect*
Query Service	*defines how applications can query databases*
Relationship Service	*for the management of references between entities*

continued on next page

continued from previous page

Name	**Goals**
Security Service	*offers support for access control, auditing, authentication and policy implementation, in order to achieve confidentiality, integrity, accountability, availability and nonrepudiation*
Time Service	*provide current time together with an error estimate, ascertain the order in which "event" occurred, generate time-based events based on timers and alarms, compute the interval between two events*
Trading Object Service	*like naming service, helps client finding the right service object*
Transaction Service	*for handling exceptions and recovering a consistent state*

- CORBA Facilities Specifications

Name	**Goals**
Internationalization and Time	*to help developers make their software independent of languages and time zones, by means of localized formatters, sorters and pattern matching methods*
Mobile Agent Facility	*to achieve a certain degree of interoperability between mobile agent platforms of different manufacturers*

- CORBA Business Specifications
    1. Task and Session
    2. Workflow Management
- CORBA Finance Specifications
    1. Currency
- CORBA Manufacturing Specifications
    1. Distributed Simulation Systems Specification
    2. Product Data Management Enablers Specification
- CORBA Medical (CORBAmed) Specifications
    1. Person Identification Service Specification
    2. Lexicon Query Service Specification

- CORBA Telecoms Specifications
  1. Audio/Video Streams
  2. CORBA/TMN Interworking
  3. Notification Service
  4. Telecoms Log Service
- CORBA Transportation Specifications
  1. Air Traffic Control

Libraries implemented according to these standards can be linked to applications and accessed either by remote method invocation, or by messaging. By reusing the objects that implement the various aspects, the development of distributed applications becomes much easier than working at the level of sockets and messages. The number of aspects covered by these specifications is remarkable, and since CORBA has been around since about 10 years, there are already many implementations of some of the specifications. They are provided both by commercial and by open-source organizations, this could guarantee long-term availability and robust systems.

The main risk of this approach is of course that the protocols are not suitable to some situations, or that the different implementations are not implemented in a perfectly compatible way. Since ORBs try to hide the network and to make programming easier by letting remotely located objects interact as if they were on the same host, there is also a risk that the underlying network disturbs the communication in some situations (§ 1.5.2, p. 31). Thus our architecture based on mobile code offers probably a better control on distribution and some optimizations.

Another problem that is sometimes mentioned is that the ORB is a big piece of software that is not able to run on devices with limited memory. To accomodate such appliances, it is necessary to write additional "gateways" that run in the service provider's environment. This requires collaboration between the service provider and customers, while the mobile code architectures avoid such collaborations.

The last shortcoming, that makes CORBA suitable for closed environments like an enterprise's intranet, but not for the open Internet is that it has no resource control or protection against denial of service attacks, like Java. The Security Service is probably quite efficient for authentication and access control, but it is improbable that CORBA will be enhanced with convenient resource control mechanisms, since it has to cope with programs written in different languages and running on different operating systems and hardware architectures.

4.7 Web technologies

There is currently an enormous amount of activity in the domain of Web-based technologies, principally XML, with new protocols being defined at an amazing rate. This domain has been fueled by the initial successes of the Web as a way to share documents and information and the extremely high commercial pressure to build distributed systems quickly and cheaply. Clearly, this development is impressive, and some people who have been working in the domain of distributed and open systems for a long time even fear that it may prevent the deployment of more mature agent technologies, which result from many years of research (Milojicic, 1999).

The basic assumption in this domain is that every site offering services to the Web can implement them, internally, using XML[10] interfaces. When a client that requests this information is a browser, the XML can be rendered into HTML, using style sheets or XSL[11] transformations. And when the client is a program, the XML can be delivered directly to that program for further processing. Of course, distributed programming should not be more complicated than that, and there are some spectaculary examples that these technologies can sometimes work as advertised. However, we have already explained in some detail why distributed programming is not that easy, and our daily experience with the Web rather shows us an abundance of invalid information, broken links and unreachable sites. So we should rather be cautious with this technology that mainly progresses by trying to fix the inherent flaws of existing systems—HTML mixes content and presentation, let's try XML!—rather than carefully building on decades of experience with distributed systems. We do agree that XML parsers, and the possibility for a human to read XML files can be useful. However, we're not at all certain that distributed systems should be build with this technology for several reasons:

1. Even if information encoded in an XML message can easily be verified using a DTD[12], and that the availability of the DTD enables automatic tools for visualizing or processing the corresponding syntactic tree, such messages still belong to the world of PDU based communication. Hence, dealing with the flow of information requires sophisticated programming.

2. Huge efforts are necessary to define good protocols (DTDs), to make them adopted and to avoid that they become obsolete. With our ap-

[10]eXtensible Markup Language, a format for the representation of complex data and document as text files.

[11]eXtensible Stylesheet Language, a language to encode transformation rules and formatting rules for XML documents.

[12]Document Type Definition, describes the permitted structure for a given type of XML document.

proach based on objects, there are also protocols, but in addition, there is encapsulation and it is possible to change completely the internals of an object, but to continue offering the same set of public methods. Because these methods can perform the necessary conversions they allow that a new version of the object remains compatible with existing clients. In the case of XML, when a client expects information formatted according to a given DTD, it is not possible to change it to DTD' and to format new information according to DTD' because all the old clients cannot read the new information. It is possible to have an XSL conversion for adapting new information to the original DTD, but the server probably doesn't know when to perform the conversion. The "mobile code" approach would be to give the XSL to the client and let it apply the transformation when needed.

3. XML technology applied naively as a successor of HTML tends to mix the aspects of persistence (how the documents are stored), communication and processing. Furthermore, they are based on unrealistic expectations of a reliable and performant network. They are also not encouraging a good control of events and system dynamicity, they rather rely on polling to know when something has changed or disappeared; this technique is not sufficient to retrieve information that has moved during reorganizations and is sensitive to delays.

4. Unlike an object, a document that gets transferred between two hosts is not able to open a socket back to the server, to process information, or to wait for events. Mobile objects are fundamentally more powerful.

Our architecture relies on Web technologies, but is not based upon them. They are convenient ways to communicate with the user using only a thin client, and we can use them when information must be "externalized" in a readable format. However, traditional entity-relation models, implemented either with objects, relational databases or both, remain at the core of our architecture, because they do have advantages.

In the end, the choice of a technology should always be made according to the needs which must be satisfied: Web technologies are probably cheaper and may require less efforts but they imply higher risks of failures since they don't handle all non-functional aspects perfectly; choosing a more complex technology certainly involves a bigger effort an may cost more time or money, but guarantees that a wider range of functional and non-functional requirements can be taken into account.

Part II
Case study

In this second part of the thesis we illustrate and justify the architectural principles of the first part. We conduct an extensive study of the “meeting scheduling” problem. We provide a new computerized solution of this concrete decision problem. Then, we proceed with a detailed presentation of the implementation.

Chapter 5

Meeting scheduling problem

Chapter highlights

- The concrete motivation to tackle this problem: meeting scheduling is time consuming and there were no Web based tools when the case study was started (summer 1999).

- Definition and example of the problem: a decision process, that must take constraints into account, and that requires communication and storage of information.

- Two fundamental assumptions in our approach: (1) full automation is too difficult to achieve and not required, (2) end users already have agendas and don't want to store their schedules on the system.

- The problem is a DEDIS, hence not too simple for a case study: **distribution** and heterogeneity result from the habits of users and the distances between them; non-trivial computations are required to take into account new activities, we must deal with change in the constraints and potential conflicts (**dynamicity**); **evolution** is desirable because people's personal and professional environments change during time, as well as the software and hardware they use.

5.1 Definition of the problem

For our case study, we have chosen the problem of meeting scheduling. We provide a new computerized service that lets a group of people collaborate in order to determine the temporal interval during which a forthcoming activity will take place.

Without appropriate tools, this is usually a time consuming task, due to the delays in communication and difficulties of collaboration, as stated in (Cesta et al., 1998):

> *Scheduling meetings for a group of users involves a high number of actors and requires a massive organizational effort, complex negotiation strategies and a huge number of communication acts, e.g., email messages, phone calls, faxes, etc. Moreover it is also necessary to find a compromise among the different users' constraints, the availability of the resources and the need of satisfying the highest number of people.*

Our tool named Meety[1], which lets each user keep his favourite agenda, exploits Web and email technologies, in order to reduce the organizational effort and time spent to schedule meetings.

Before we describe the characteristics of the meeting scheduling problem, the existing solutions, and the benefits of our new approach, let us define it more precisely. First of all, scheduling an event (a meeting or an activity) implies a **decision process** in which it is necessary to determine **when** the event should take place. In many cases, it is not possible to decide blindly as a set of **constraints** must be taken into account. In addition, this decision must not be forgotten but it must be **recorded** and **communicated** by appropriate means to all concerned parties. Our work is focused on this decision process and it will be our definition of the meeting scheduling problem, even if scheduling a meeting actually involves many other decisions about people, activities and facilities.

5.2 Example

We can illustrate this decision process with the example of a quartet of musicians who don't gather on a regular basis but want to attempt to play together. In this particular case, the scheduling problem is to decide when the session will take place.

Even in this simple case, the constraints that must be taken into account are quite complex. Presumably, the four players already have a lot of other

[1]http://meety.unige.ch/

scheduled activities that reduce their availability. In scheduling terms, these other activities are constraints that eliminate many dates and times (when the activity cannot be shifted or cancelled), or make some solutions more desirable than others (it would be possible to schedule the session just before a performance, but it's probably not the best time). Our four musicians must also consider the location: availability of rooms, times to travel to this location, etc. This is a second set of constraints that may impact on the decision in subtle ways. Additional constraints may be expected: the duration of the session, the time necessary to obtain an uncommon score, etc.

The decision itself can be made in several ways. A single person may collect all necessary informations and make the decision alone, then communicate it to others. The decision can also be a consensus, emerging from a collective process between several people. It may also be the result of an elaborate computation.

In all cases, the four participants must be informed of the decision, and remember it, otherwise the event cannot occur. We can expect that each of them uses an **agenda** to record these informations (but we cannot assume that it is an electronic device). It is also necessary to ensure that all **resources** needed for the session (e.g. the room, instruments, seats) will be available. This resource reservation, or resource allocation process frequently accompanies scheduling problems. In this case, its solution can also be based on some kind of agenda.

5.3 Partial automation

The goal of the Meety system—the Web service developed for our case study—is to support decision processes similar to the one described in the previous section, where a complex set of constraints must be taken into account. It performs a partial treatment of the meeting scheduling problem, where communications and the representation of constraints are made easier by the computer, but where the final decision is left to the people using it.

For this case study, it was not conceivable to fully automate the resolution of the problem, which is far too complex in an unrestricted setting. Our intent was rather to provide a service that would be simple to understand and easy to use, in order to assist a large number of people and provide them an opportunity of saving time.

Other researchers (Cesta et al., 1998; Sen and Durfee, 1996) have tried to fully automate the decision process. They have developed sophisticated programs that they call **agents** which are able to make the decisions on behalf of a user. These agents negotiate with other agents, or other people, following elaborate protocols, in order to manage agendas without intervention of their owners (§ 7.4.2, p. 129 and next ones). It requires lots of efforts and of patience to represent the subtle factors (among other the social ones) that may be relevant in this process. Without a very cautious programming and an extensive knowledge of the user's context and habits, such agents are not able to take the right decisions, and their behavior cannot satisfy the user. We supposed that people using our system to schedule meetings would be reluctant to spend time configuring their agent and we preferred to skip this part of the automation. However, it is not unthinkable to add it later, since the system is structured according to the architecture presented in Chap. 3 and thus easily extensible using mobile code.

Another particularity of our approach to the meeting scheduling problem is that it is only loosely connected to the agendas of the users, which are not part of Meety. In the general case, people involved in the decision process do not necessarily attend the event. Thus the meeting scheduling problem cannot be reduced to finding a "free" interval in these peoples' agendas. On the other hand it was very unlikely that everyone would agree to store his calendar in our system, thus we abandoned this idea. One benign consequence is that the users may end up repeating some operations several times (rejecting a busy time for several meetings).

5.4 Problem characteristics

The three characteristics of DEDIS (dynamicity, evolution, distribution) are well represented in the meeting scheduling problem, making it an interesting candidate for our case study.

The most obvious one is **distribution**: we all have our own occupations, which are very different from those of our colleagues and close relatives, and the way we manage our time is something very personal. Some of us use a pesonal computer to manage our agenda, or a portable electronic device, some prefer to have it written on paper or on a wall calendar, or even to rely on their good memory. Consequently, it is very important that systems which try to solve this problem are able to take into account this **heterogeneity** of means, which has been reported by (Coleman, 1999)[2]:

> *Interestingly, in a recent survey of 611 Internet users who use a calendar, 50% were still using paper calendars. The other 50% break out into a variety of different schedulers and PIMs [Personal Information Managers] detailed in the figure. Internet calendars have not yet caught on. Of the 87 heavy schedulers surveyed, 85% had never heard or tried them. Of the 15% that did most of those*

[2]The original figure is reproduced as Fig. 5.1, p. 96.

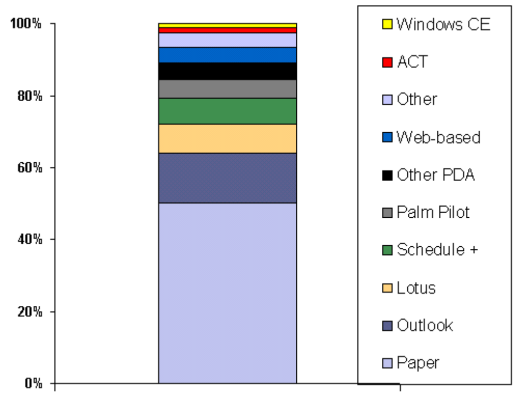


Figure 5.1: Calendar choices of Web users (Coleman, 1999). N=611, data gathered in April 1999, compiled by NFO interactive. See § 7.1, p. 112 for an overview of these products.

> *felt that they did not meet their needs. A few percent use Internet calendars to track birthdays, but none surveyed used it to share data with others.*

The second facet of distribution is that everyone is frequently changing location and doesn't meet other people everyday. Thus the second requirement is that the system facilitates collecting the information required for a good decision. It should essentially reduce communication lags and avoid ambiguities.

The **dynamicity** characteristic is also very present. People are frequently adding new activities to their schedules, and each new activity reduces the possibilities for the other ones, since it is usually not possible to do two things at the same time. For the meeting scheduling system, this means that there will never be a definitive solution, but that changes must be continuously taken into account. It is necessary to reflect new constraints or decisions in the system's state, and to check that these new elements don't conflict with earlier choices.

Having an activity that prevents another one to take place during a certain period does not mean that a conflict happens. The necessity to explore a potentially large space, searching for a good solution, is in the nature of the meeting scheduling problem. Constraints between activities just mean that some parts of this space contain no solution. However, this can degenerate in two embarassing cases:

- When it is not possible to allocate a time stretch to an activity because there are too many constraints (e.g. the meeting participants are too busy). In this case, it is necessary to **relax** some of these constraints, for instance by postponing the activity, or by cancelling another one to create an opportunity. This situation can be tricky, but unlike the second case, there is no risk to bring the system into an incoherent state as long as the activity doesn't receive an interval that overlaps with other activities.

- The "conflicts through shared resources" (Sen and Durfee, 1996, p. 10) occur when a single resource (or person) is required by two actvities that could potentially take place in overlapping time intervals. If these two activities are scheduled concurrently, the two decision processes may result in assigning colliding time windows to the two activities. In this case, the system state would be inconsistent. A simple illustrative instance is when you need to meet two people separately during the next day, and both tell you that they have set up everything to meet you from 11:00 to 12:00.

These conflicts are inherent to the meeting scheduling problem and contribute to its difficulty. Several techniques can be applied to solve them, that we will present in § 7.4.3, p. 132.

The **evolution** characteristic is present as well. We think that it is of utmost importance for meeting scheduling software to be able to evolve with communication technologies, like the currently exploding market of mobile devices, or future versions of the Internet protocols. It is also essential that such systems can be easily associated with new applications, for instance a system for room reservations, or new personal calendaring services. In (Grudin and Palen, 1995) this need to integrate various tasks is illustrated by:

> *Conference room availability was sometimes described as the most critical aspect of scheduling a meeting, so the ability to schedule rooms or equipment through the system is another example of a feature that promotes use.*

Additionally, meeting scheduling is a typical application where the needs of the users are very difficult to foresee, since the management of one's time is so personal, and the dependence on the context is so strong. For this reason, we can expect that each user will have his own requirements, rules and preferences and that they will change along with his environment (e.g. working or familial situation). Thus it is critical to be prepared to satisfy these requirements, even if they are completely unexpected or conflict with other ways of working or of thinking.

One of the goals of this thesis is to show that using mobile code provides a smooth evolution path for services with such requirements. We believe that our architecture, where new rules and behaviors can be added to the service at any time by independent "extension programmers", offers the necessary adaptability. We do not expect each user to write his own extensions, but it is thinkable to have a public repository where these extensions can be stored, retreived and activated on demand. Some useful rules or automations are those which increase the coherence of information, propagate changes to other systems, or prevent errors and forgetfulness. Interesting examples could be: additional notifications for the people involved in the organization of an event, automatic creation of a new meeting for recurrent events, automatic answer for some specific periods, etc. A whole chapter is devoted to these extensions (Chap. 9).

Chapter 6

Meety use-cases

Chapter highlights

- A comprehensive overview of Meety's features: for readers who have never seen the service, the use-case diagram of Fig. 6.1, p. 100 graphically depicts the main interactions occuring between the service and its users.

- Detailed scenarios: use-cases and screen captures describe the intended function of the service and possible interaction.

- The limitations of the service: because we knew that a fully automatic tool would be too difficult to develop, and probably not suited to a large number of users, and also because of our preference for small services that can be easily extended or integrated (Chap. 3), the functionality of Meety is intentionally limited to the communication of constraints, and the persistent storage of meeting informations.

- A truthful description: all what is described here is currently implemented and can be used by anyone who needs a good tool for date selection!

6.1 Obtain password

Intent: This step is the basis of our authentication and access control policy (§ 3.4.4, p. 63). Since it interacts only with registered users, the system can:

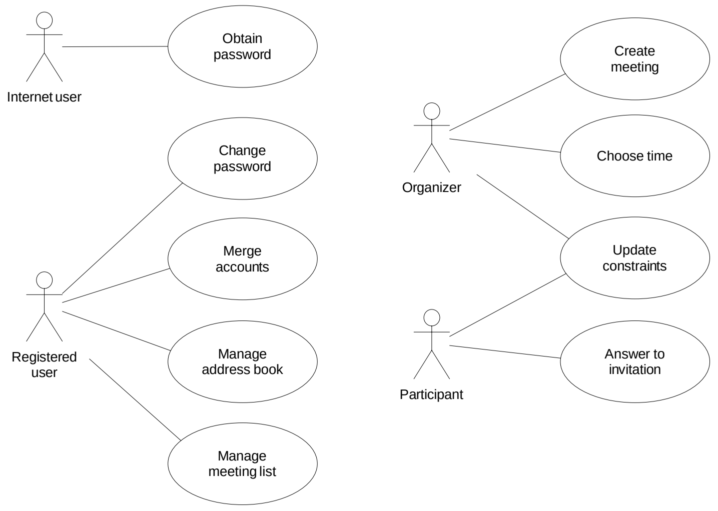
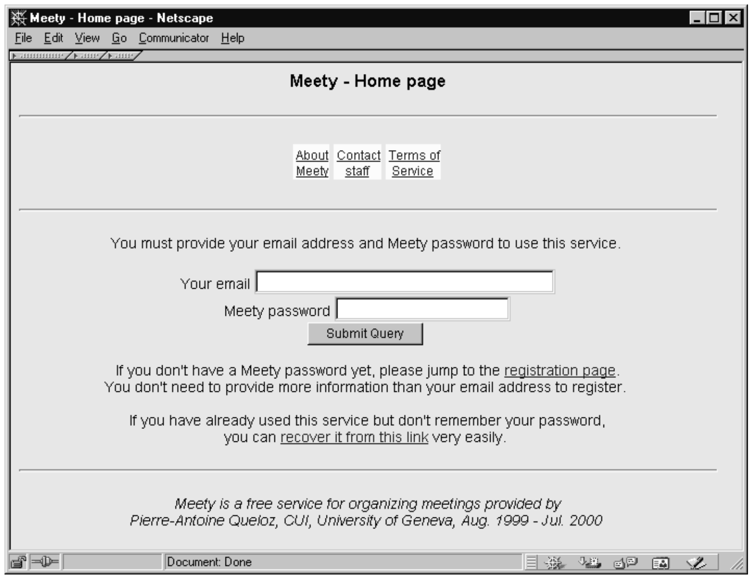


Figure 6.1: Overview of the main operations offered by the service. The stick figures represent actors (different roles played by people or software agents when they interact with the system). Each use-case is rendered as an ellipse. An *Organizer* must be a *Registered user* (possess a password). A *Participant* may interact with the system without providing a password.

1. prevent disclosure of meeting informations to people who are not explicitly invited in the scheduling process, and prevent modification of meeting data and of answers by unauthorized users;
2. ensure that people are not using fake email addresses; this is important, since emails are sent on their behalf in the scheduling process, and without authentication, it would be easy to pretend being someone else; for instance, anybody in a department could use the director's email address to summon a meeting, or illegitimately answer instead of him.

It is also important from a legal point of view, since people must accept the terms of service before using Meety.

Main flow of events: (1) The *Internet user* requests the “home” page of the service (Fig. 6.2, p. 101). (2) The *Internet user* requests the “registration” page. (3) The *Internet user* enters his email address and clicks on “submit”. (4) The system displays the terms of service. (5) The *Internet user* clicks on button [accept]. (6) The system generates a random pass-


Meety - Home page - Netscape
File Edit View Go Communicator Help
Submit Query
Document: Done

Meety - Home page

About Meety | Contact staff | Terms of Service

You must provide your email address and Meety password to use this service.

Your email

Meety password

If you don't have a Meety password yet, please jump to the registration page. You don't need to provide more information than your email address to register.

If you have already used this service but don't remember your password, you can recover it from this link very easily.

Meety is a free service for organizing meetings provided by Pierre-Antoine Queloz, CUI, University of Geneva, Aug. 1999 - Jul. 2000

Figure 6.2: The "home" page of the service.

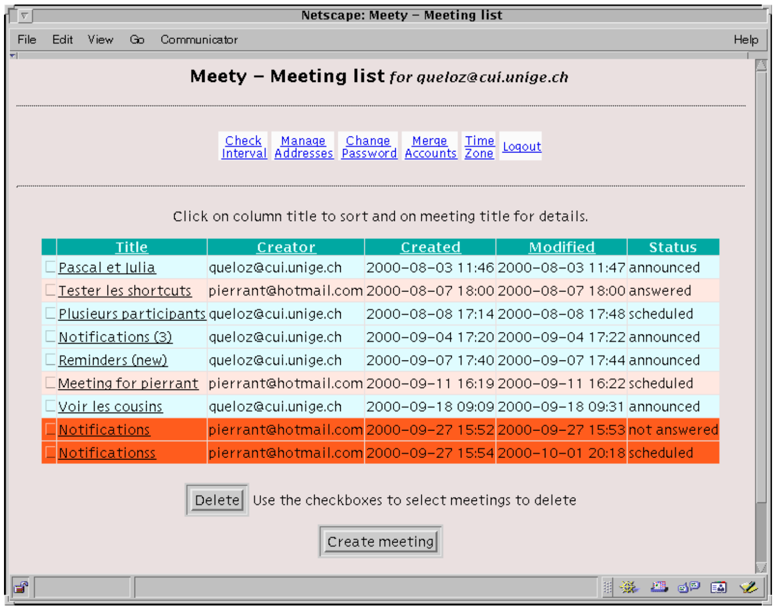
word and sends it inside an email message to the address provided by the user. (7) The system displays a welcome page. (8) The *Internet user* receives the message containing the password and becomes a *registered user*.

Exceptional flow of events: The *Internet user* does not accept our terms of service at (5). The system doesn't generate a password and displays an informative page.

Exceptional flow of events: The address provided by the *Internet user* at step (3) is not correct. The message is lost or reaches the real owner of the email address and the *Internet user* doesn't know the password. In this last case, this unexpected message may cause some surprise but no prejudice.

Exceptional flow of events: The address provided by the *Internet user* at step (3) is already associated to a password. The system doesn't display the terms of service, it displays an informative page and sends an email message containing the current password to the given address.

We found that this simple authentication strategy is efficient and simple to understand. It doesn't take more than a couple of minutes to accomplish



Meety – Meeting list *for queloz@cui.unige.ch*

Click on column title to sort and on meeting title for details.

Title	Creator	Created	Modified	Status
☐ Pascal et Julia	queloz@cui.unige.ch	2000-08-03 11:46	2000-08-03 11:47	announced
☐ Tester les shortcuts	pierrant@hotmail.com	2000-08-07 18:00	2000-08-07 18:00	answered
☐ Plusieurs participants	queloz@cui.unige.ch	2000-08-08 17:14	2000-08-08 17:48	scheduled
☐ Notifications (3)	queloz@cui.unige.ch	2000-09-04 17:20	2000-09-04 17:22	announced
☐ Reminders (new)	queloz@cui.unige.ch	2000-09-07 17:40	2000-09-07 17:44	announced
☐ Meeting for pierrant	pierrant@hotmail.com	2000-09-11 16:19	2000-09-11 16:22	scheduled
☐ Voir les cousins	queloz@cui.unige.ch	2000-09-18 09:09	2000-09-18 09:31	announced
☐ Notifications	pierrant@hotmail.com	2000-09-27 15:52	2000-09-27 15:53	not answered
☐ Notificationss	pierrant@hotmail.com	2000-09-27 15:54	2000-10-01 20:18	scheduled

Use the checkboxes to select meetings to delete

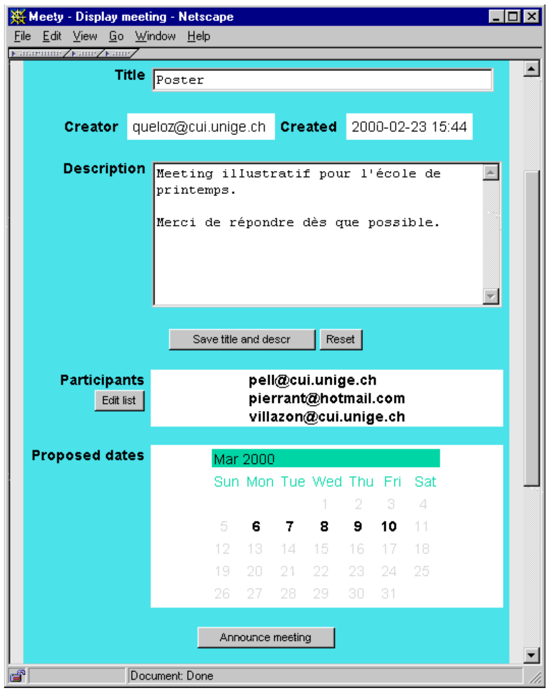
Figure 6.3: The "main" page of the service, displayed when the user has logged in. It shows the current list of meetings for this user, and provides links to the main actions. Colors are used to indicate the status of meetings, and the list can be sorted according to different keys, by clicking on the column titles. The details of meetings are displayed when the user clicks on titles.

this first step, which also includes accepting our terms of service, and we had no complaints from our users.

6.2 Manage meeting list

Intent: Let the user visualize and manage his meeting list.

Main flow of events: (1) The *Registered user* requests the "home" page of the service. (2) The *Registered user* enters his email address and his current password. (3) The system checks the validity of this password. (4) The system displays the "main" page of the service (Fig. 6.3, p. 102). Meeting titles and attributes are displayed in a table. Controls are provided to sort the list according to several criteria, to delete meetings, to create meetings, and to access other functions (other use-cases). (5) The user follows the "Logout" link to terminate the session.



Title: Poster

Creator: queloz@cui.unige.ch Created: 2000-02-23 15:44

Description:

```
Meeting illustratif pour l'école de
printemps.

Merci de répondre dès que possible.
```

Participants: pell@cui.unige.ch, pierrant@hotmail.com, villazon@cui.unige.ch

Proposed dates:

Mar 2000

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	**6**	**7**	**8**	**9**	**10**	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Figure 6.4: Page filled by *Organizer* to announce a new meeting.

Exceptional flow of events: If the user is not able to provide the right password at (2), the use-case is aborted. The system displays an appropriate result page.

Exceptional flow of events: If the user does not terminate the session at (5) the system automatically closes it after 60 minutes of inactivity.

6.3 Create meeting

Intent: Define the attributes of a new meeting (title, description, possible dates) and send an invitation to participants.

Main flow of events: (1) The *Organizer* requests the "home" page of the service. (2) The *Organizer* enters his email address and his current password. (3) The system checks the validity of this password. (4) The system displays the "main" page of the service. (5) The *Organizer* clicks on button [Create meeting]. (6) The system adds a new meeting to the list. (7) The *Organizer* clicks on the title of the new meeting. (8) The systems displays the details of the new meeting (Fig. 6.4, p. 103). (9) The *Organizer* edits the title of the meeting. This title is displayed as the subject of the related email messages and in the meeting list of the main page. (10) The *Organizer* edits the description of the meeting. This description is copied in the body of email messages sent to participants. (11) The *Organizer* clicks on button [Save title and descr]. (12) The system stores the new title and description. (13) The *Organizer* clicks on button [Edit list] to select the meeting participants (that will receive the announcement). (14) The *Organizer* clicks on button [Edit dates] (not visible on Fig. 6.4, p. 103) to select an initial set of days, and optionally hours. He must check his own agenda, or any other calendar relevant to the event, in order to provide a consistent set of time intervals. (15) The *Organizer* can select for which events he wants to be notified (not visible on Fig. 6.4, p. 103). (16) The *Organizer* clicks on button [Announce meeting]. (17) The system checks that there is at least one participant in the list that has not yet received an announce for this meeting (no duplicates are sent). (18) The system checks that at least one date is proposed. (19) The system sends an email message to all participants. This message contains the meeting title and description. It invites the participant to follow a hyperlink that leads to a page where the participant can give his answer.

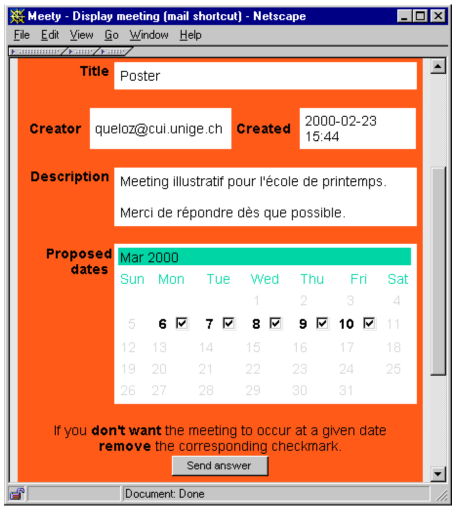
Exceptional flow of events: If the sequence is interrupted before (11) the title and description will not be saved, but the meeting will remain in the *Organizer*'s meeting list.

Exceptional flow of events: If the sequence is interrupted after (11) the state of all attributes of the meeting will be kept in the system, and the meeting will remain in the meeting list.

Exceptional flow of events: Events (9)-(15) are optional and can occur in a different order. The only limitation is that changes in the title and description are not stored if (11) is omitted, and that the invitations will not be sent if (17) or (18) fails.

6.4 Answer to invitation

Intent: Gather constraints from the participants for a given meeting. The participants can indicate when the meeting should take place, accord-


Meety - Display meeting (mail shortcut) - Netscape
File Edit View Go Window Help
Send answer
Document: Done

Title	Poster
Creator	queloz@cui.unige.ch
Created	2000-02-23 15:44
Description	Meeting illustratif pour l'école de printemps. Merci de répondre dès que possible.

Proposed dates: Mar 2000

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6 ✓	7 ✓	8 ✓	9 ✓	10 ✓	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

If you **don't want** the meeting to occur at a given date **remove** the corresponding checkmark.

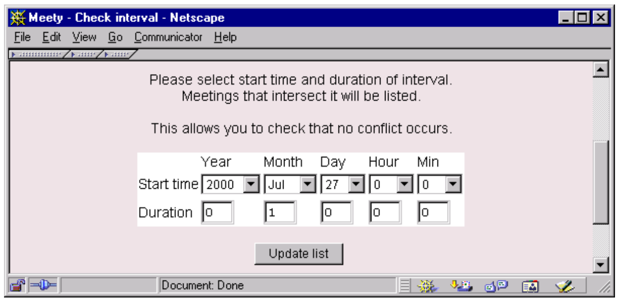
Figure 6.5: Page displayed to gather answers from participants. Checkboxes are used to deselect dates where the meeting should not be scheduled.

ing to their availabilities and preferences, but without revealing private informations.

Main flow of events: (1) The *Participant* receives the invitation by email. (2) The *Participant* sends an HTTP request to the server, using his browser and the URL included in the invitation. (3) The system checks that the URL is valid, no password is necessary to provide an answer. (4) The system generates a formular for the answer (Fig. 6.5, p. 105). (5) The *Participant* checks his availability in his personal agenda, or others calendars that may contain information relevant to the event, and deselects dates or times where the meeting should not be scheduled. (6) The *Participant* types a comment in the text area (not visible in the figure). (7) The *Participant* clicks on button [Send answer]. (8) The system stores the answer and the comment, then it displays an informative page.

Exceptional flow of events: Instead of (1)-(3) the *Participant* sees a new meeting in his meeting list. If he clicks on the meeting title, he can provide an answer according to the sequence of events (4)-(8).

Exceptional flow of events: Events (5) and (6) are optional and their relative order is not important. But they must occur before (7).


Meety - Check interval - Netscape
File Edit View Go Communicator Help
Document: Done

Please select start time and duration of interval.
Meetings that intersect it will be listed.

This allows you to check that no conflict occurs.

	Year	Month	Day	Hour	Min
Start time	2000	Jul	27	0	0
Duration	0	1	0	0	0

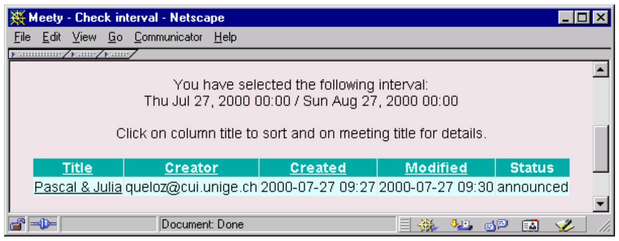
Update list

Figure 6.6: This small form is displayed when a *Registered user* clicks on "Check Interval".

6.5 Update constraints

Intent: Make the detection of potential conflicts and the modification of constraints easier. This use-case typically starts when the user reserves a new time stretch in his agenda and suspects that this new reservation overlaps the domains of other meetings.

Main flow of events: (1) The *Registered user* requests the "home" page of the service. (2) The *Registered user* enters his email address and his current password. (3) The system checks the validity of this password. (4) The system displays the "main" page of the service. (5) The *Registered user* clicks on button [Check Interval]. (6) The system displays a form for the selection of a temporal interval (Fig. 6.6, p. 106). (7) The *Registered user* selects the start time and the duration of the interval and clicks on button [Update list]. (8) The system displays a subset of

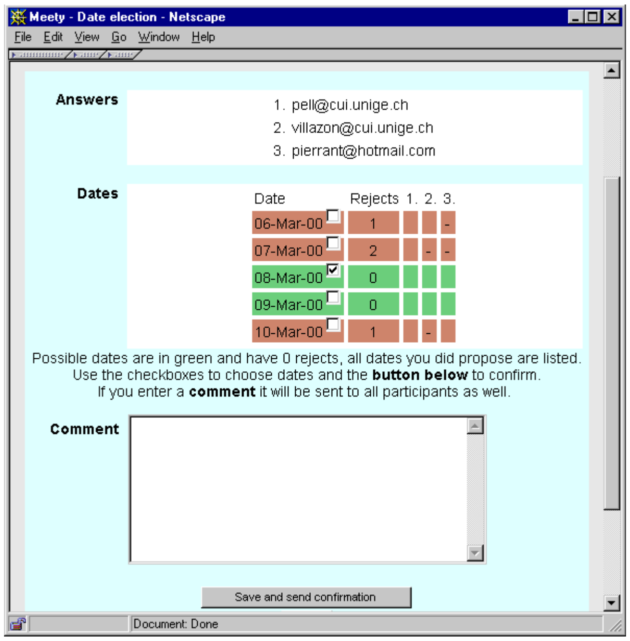

Meety - Check interval - Netscape
File Edit View Go Communicator Help
Document: Done

You have selected the following interval:
Thu Jul 27, 2000 00:00 / Sun Aug 27, 2000 00:00

Click on column title to sort and on meeting title for details.

Title	Creator	Created	Modified	Status
Pascal & Julia	queloz@cui.unige.ch	2000-07-27 09:27	2000-07-27 09:30	announced

Figure 6.7: A subset of meetings is displayed when the *Registered user* has specified a temporal interval. Only those meetings that have possible dates intersecting the interval are displayed.


Meety - Date election - Netscape
File Edit View Go Window Help
Save and send confirmation
Document: Done

Answers

1. pell@cui.unige.ch
2. villazon@cui.unige.ch
3. pierrant@hotmail.com

Dates

Date	Rejects	1.	2.	3.
06-Mar-00 ☐	1	🟥	🟥	-
07-Mar-00 ☐	2	🟥	-	-
08-Mar-00 ☑	0	🟩	🟩	🟩
09-Mar-00 ☐	0	🟩	🟩	🟩
10-Mar-00 ☐	1	🟥	-	🟥

Possible dates are in green and have 0 rejects, all dates you did propose are listed.
Use the checkboxes to choose dates and the **button below** to confirm.
If you enter a **comment** it will be sent to all participants as well.

Comment

Figure 6.8: Page used by the *Organizer* to schedule the meeting according to participants' answers.

meetings for this user that intersect the given interval (Fig. 6.7, p. 106). (9) The user clicks on the meeting titles to check that there is no conflict in his propositions or answers. (10) The user modifies his propositions or answers if necessary.

Alternate flow of events: A *Participant* that has not obtained a password is not able to use this feature, but he is able to visit the URL provided by the system more than once, in order to update his answer.

6.6 Choose time

Intent: Provide an understandable view of all the constraints given by participants; let the *Organizer* schedule the meeting and send confirmations to participants.

Main flow of events: (1) The *Organizer* requests the "home" page of the service. (2) The *Organizer* enters his email address and his current password. (3) The system checks the validity of this password. (4) The system displays the "main" page of the service. (5) The *Organizer* clicks on a meeting's title. (6) The system displays the meeting's details. (7) The *Organizer* clicks on button [Choose now] in the section "Elected dates". (8) The system displays the proposed dates and the answers of participants (Fig. 6.8, p. 107). Dates that have been rejected by participants have a red background, dates that have not been rejected have a green background. (9) The *Organizer* selects the best time for the meeting, according to the given constraints and comments. (10) The *Organizer* enters a comment in the corresponding text area. (11) The *Organizer* clicks on the button [Save] button. (12) The system stores the choice of the organizer, then it sends an confirmation message to all participants. The comment of (10) is inserted into the body of the message, as well as the chosen date.

Exceptional flow of events: Event (10) is optional.

If all dates have been rejected by at least one participant, the problem is over-constrained. While this situation would be very tricky for a fully-automated scheduling system, the *Organizer* still has several possibilities: sending a new invitation with new dates, overlooking the preferences of some participants, calling some participants to negotiate with them, etc. Our opinion is that the final decision is so much dependent on a large and complex context and on social factors that it's not reasonable to try to automate it. Nevertheless, the second column of the "Dates" table (cumulated rejects) gives a very good indication of where an imperfect solution should be looked for.

When participants receive the confirmation, they can update their personal agendas accordingly. At this stage, it may be necessary to update the constraints of other meetings (§ 6.5, p. 106) in order to avoid conflicts through shared resources.

6.7 Change password

Intent: A *Registered user* chooses a password that is easier to remember than the long random password provided by the system.

Main flow of events: (1) The *Registered user* requests the "home" page of the service. (2) The *Registered user* enters his email address and his current password. (3) The system checks the validity of this password and displays the "main" page. (4) The *Registered user* requests the "change password" page. (5) The *Registered user* provides his current password and the new password. (6) The system checks that the first password is correct and that the new password is acceptable. (7) The system stores the new password. (8) The system displays a result page.

Exceptional flow of events: If one of the passwords is not acceptable at (6), the system displays an appropriate error message and asks the user to retry point (5).

The current password is necessary at point (5) to avoid that someone else changes the password when the user leaves the terminal without ending the session.

6.8 Merge accounts

Intent: Transfer meetings from one account to another account. This is necessary because email addresses are used to identify accounts and people can have more than one address.

Main flow of events: (1) The *Registered user* requests the "home" page of the service. (2) The *Registered user* enters the email address he wants to keep and the corresponding password. (3) The system checks the validity of this password and displays the "main" page. (4) The *Registered user* requests the "merge accounts" page. (5) The *Registered user* provides his current password, the address he wants to erase and the corresponding password. (6) The system checks that both passwords are correct. (7) The system transfers all meetings from the second account to the current one. (8) The system displays a result page.

Exceptional flow of events: If one of the passwords is not acceptable at (6), the system displays an appropriate error message and asks the user to retry point (5).

6.9 Manage address book

Intent: Let each user store and manage an address book in the service. The addresses of meeting participants are selected in the address book. Each address book entry has four fields: nickname, first name, last name and email address.

Main flow of events: (1) The *Registered user* requests the "home" page of the service. (2) The *Registered user* enters his email address and his current password. (3) The system checks the validity of this password and displays the "main" page. (4) The *Registered user* requests the

"Manage Addresses" page. (5) The system displays the user's address book in a table and provides usual management functions: sorting the list alphabetically according to one of the fields; creating, deleting, or modifying an entry; importing an existing list (from Netscape, Hotmail, etc.), exporting the list in a standard format.

To a large extent, the address book functionalities are independent from the meeting scheduling application and could be reused for other Web services. A detailed description can be found in (Greppin, 2000).

Chapter summary

This chapter gives a detailed description of the system and should have convinced the reader that the system has all the required functionalities for the quick organization of meetings involving a large number of participants. In the next chapter, we compare our solution with other systems that allow to perform similar tasks.

Chapter 7

Software for meeting scheduling

Chapter highlights

- What distinguishes Meety from commercial desktop products: we review several such products and describe their main shortcomings (1) lack of inter-operability—even if standardization efforts are underway, (2) reliance on shared calendars—requires a high commitment from all users.

- What distinguishes Meety from other Web services for scheduling: we review several such services and compare their features with those of Meety. The main differences found are (1) the appearance—Meety is much more sober, (2) the width of application—Meety is focused on date selection, (3) the ease of use—Meety has some features that others are lacking, (4) the technology—Meety is the only one that can be extended with mobile code.

- Research in distributed constraint satisfaction: we provide a detailed survey of theoretical work related to our case study, such that the reader can grasp our approach and our implementation.

- Scheduling can be formalized as a constraint satisfaction problem: there are several approaches to search solutions, characterized by the exponential growth of the search space when variables are added and the difficulty to avoid local optima when some solutions are more desirable than others.

- Other methods are required when the problem is distributed: when variables and constraints cannot be grouped on a single host, new approaches involving negotiation are required.
- Dynamicity and concurrency make the problem more difficult: dynamicity results in new variables and constraints that must be taken into account, concurrency can bring the system in an inconsistent state if resources are not reserved carefully.
- Full automation of the decision process is very difficult: some researchers have shown that it is possible to capture user requirements using knowledge management or machine learning techniques, in order to parameterize a personal agent that schedules events automatically, but they would have been impossible to apply in our case.

7.1 Commercial desktop products

Many desktop products for time-management are available on the market. This section reviews some of them, from the perspective of our meeting scheduling problem. We have intentionally left out three categories of products:

1. those which offer calendaring for a single user but do not support group calendaring, like Maximizer[1] or Sharkware[2]
2. those where users share calendars for the purpose of distributing or delegating tasks, like Task Plus Professional[3]
3. those which support staff scheduling and resource allocation on a large scale and in a complex setting, e.g. for a whole department, but are not convenient for a group of people that must decide when a specific event must take place; for instance WallCHART Resource Scheduling System[4], FlexTime[5], Scheduler Plus 2001[6]

7.1.1 Sun Calendar Manager

This tool runs under OpenWindows and CDE on SparcStations, it can be used by each user in order to store appointments on his host machine. The current date is displayed when it is iconized (Fig. 7.1, p. 113).

[1]http://www.maximizer.com/
[2]http://www.sharkware.com/
[3]http://members.aol.com/contplus4/taskpro/index.htm
[4]http://www.wallchart.com/
[5]http://www.staff-scheduling.com/
[6]http://www.ceosoft.com/schedulerplus2001/index.htm

Figure 7.1: The icon of Calendar Manager.

Accessing the calendar of another user is possible when the corresponding host machine is known and reachable, and if the owner has granted access permissions. As it can be seen on Fig. 7.2, p. 114, privacy of informations is controlled by assigning to each appointment one of the three values "Show Time and Text", "Show Time Only", "Show Nothing". The user can also grant privileges to his co-workers (browse, insert, delete).

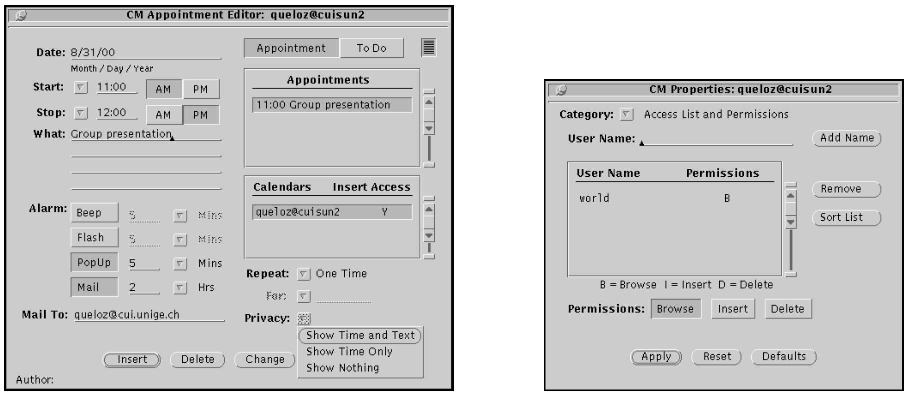
To schedule a meeting with other Calendar Manager users, it is possible to browse multiple calendars simultaneously. The calendars are "overlaid" and the more people have an appointment at a given point, the darker the shading of the corresponding area. It makes it possible to find one or more suitable time windows and to send an email to invite people. When the message is sent directly from Calendar manager, people using Sun's Mail Tool are then able to drag the appointment directly into their calendars.

7.1.2 Microsoft Outlook

Informations and images in this section were taken from Microsoft Web site[7]. This software is the successor of another widely adopted Microsoft product: Schedule+. Together, they were adopted by about 20% of the 611 users surveyed in 1999 (Fig. 5.1, p. 96). Among many other features, the user can use it in order to manage his personal calendar on his PC. Calendar information can be periodically stored on a server, from there it can be replicated (synchronized) on other computers used by the same person, or shared with other Outlook users. It is also possible to publish one's calendar as Web pages (Fig. 7.3, p. 114).

To schedule meetings, Outlook users can visualize the free and busy times of all meeting participants "overlaid" together, or they can see an expanded list of free and busy times. They can also enter a list of meeting invitees, and click AutoPick to automatically identify the next time everyone is free (Fig. 7.4, p. 115). With this feature, it is also possible to designate a set of suitable rooms and let Outlook select one of them for the corresponding stretch of time, according to the availabilities of the rooms themselves.

[7]http://www.microsoft.com/office/outlook/


CM Appointment Editor: queloz@cuisun2

Date: 8/31/00
Month / Day / Year
Start: 11:00 AM PM
Stop: 12:00 AM PM
What: Group presentation

Alarm:
Beep 5 Mins
Flash 5 Mins
PopUp 5 Mins
Mail 2 Hrs

Mail To: queloz@cui.unige.ch

Appointment | To Do

Appointments
11:00 Group presentation

Calendars	Insert Access
queloz@cuisun2	Y

Repeat: One Time
For:
Privacy:
Show Time and Text
Show Time Only
Show Nothing

Insert Delete Change

Author:

CM Properties: queloz@cuisun2

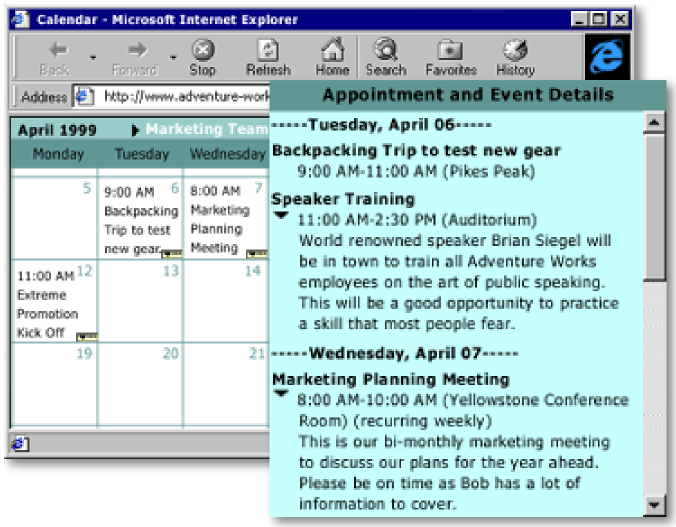
Category: Access List and Permissions
User Name:

User Name	Permissions
world	B

B = Browse I = Insert D = Delete

Permissions: Browse Insert Delete

Figure 7.2: Left: an appointment in Calendar Manager with the three possible privacy values. Right: granting privileges to co-workers.



http://www.adventure-work

April 1999 Marketing Team

Monday	Tuesday	Wednesday
5	6 9:00 AM Backpacking Trip to test new gear	7 8:00 AM Marketing Planning Meeting
12 11:00 AM Extreme Promotion Kick Off	13	14
19	20	21

Appointment and Event Details

-----Tuesday, April 06-----

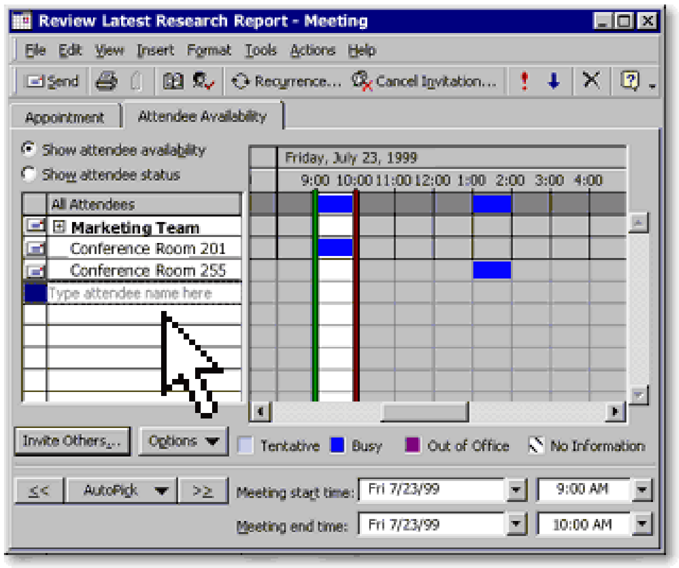
Backpacking Trip to test new gear
9:00 AM-11:00 AM (Pikes Peak)

Speaker Training
11:00 AM-2:30 PM (Auditorium)
World renowned speaker Brian Siegel will be in town to train all Adventure Works employees on the art of public speaking. This will be a good opportunity to practice a skill that most people fear.

-----Wednesday, April 07-----

Marketing Planning Meeting
8:00 AM-10:00 AM (Yellowstone Conference Room) (recurring weekly)
This is our bi-monthly marketing meeting to discuss our plans for the year ahead. Please be on time as Bob has a lot of information to cover.

Figure 7.3: Outlook calendar published on the Web.



Meeting start time: Fri 7/23/99 9:00 AM

Meeting end time: Fri 7/23/99 10:00 AM

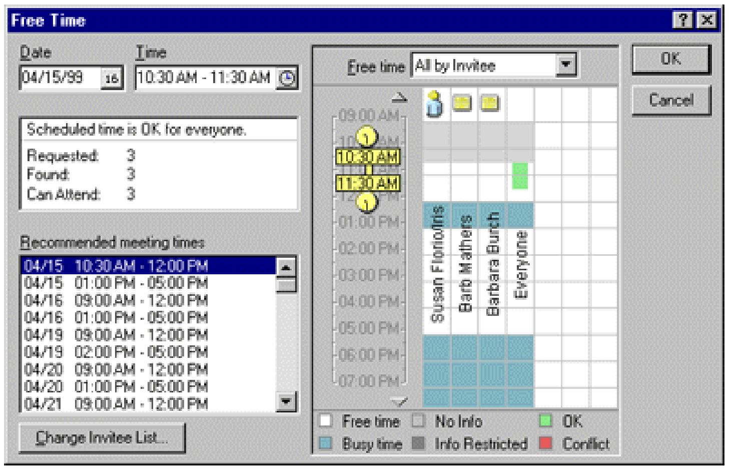
Figure 7.4: Selection of a free interval with Outlook's AutoPick feature.

When they have determined the best time for the meeting, they send an invitation message to the participants, who can either accept the invitation, reject it or indicate that they hesitate. If they accept the invitation, the new meeting is automatically added to their own calendar.

The cost of Outlook alone is approximately 100$ (2000-08-15), and even cheaper when bought with an "office suite". However, interesting features are only available with an additional "Exchange Server" which costs around 1000$.

7.1.3 Lotus Notes

Informations and images from this section were obtained from (Florio, 1998). Lotus Notes was used by approximately 8% of the users surveyed in Fig. 5.1, p. 96. Like Outlook, this product integrates calendaring and messaging and thus offers some support for meeting scheduling. Each user controls which parts of his agenda are published, and the system is able to check the availability of designated meeting participants and to suggest meeting times. In addition to the basic "accept" and "decline" operations, delivery options distinguish required, optional and "for your information" invitations, and allow counter proposals (a participant proposes another time) and delegation (a participant asks someone else to attend the meeting on his behalf).



Date: 04/15/99
Time: 10:30 AM - 11:30 AM
Free time: All by Invitee

Scheduled time is OK for everyone.
Requested: 3
Found: 3
Can Attend: 3

Recommended meeting times

04/15	10:30 AM - 12:00 PM
04/15	01:00 PM - 05:00 PM
04/16	09:00 AM - 12:00 PM
04/16	01:00 PM - 05:00 PM
04/19	09:00 AM - 12:00 PM
04/19	02:00 PM - 05:00 PM
04/20	09:00 AM - 12:00 PM
04/20	01:00 PM - 05:00 PM
04/21	09:00 AM - 12:00 PM

Figure 7.5: Selection of a free interval with Lotus Notes.

Lotus Notes sends numerous messages to support the decision process, and facilitates the transfer of new appointments from email messages into the user's calendars. And when a meeting "chair" reschedules or cancels a meeting, notifications are sent automatically.

Lotus Notes can also find rooms or resources that are available at the time of the meeting, and nominate a "resource owner" who handles policies and reservations for a particular resource.

Lotus Organizer 6.0 (79$ in August 2000) is another calendaring tool with similar meeting scheduling features. Both products allow synchronization with a Palm Pilot.

7.1.4 Meeting Maker

Meeting Maker[8] is a client/server group scheduler for Windows, Macintosh, and Unix platforms. Users can schedule meetings, keep prioritized to-do lists, and coordinate schedules with other Meeting Maker users. It offers a Java client and Palm synchronization for mobile users. Calendar information is

[8]http://www.meetingmaker6.com/

centralized on a server, thus immediately made available for scheduling purposes, without need for users to explicitly publish updates and without the lag that can occur with email-based systems. It costs $\approx$ 1000$ for 10 clients (2000-02-23).

7.1.5 Office Tracker Scheduler

Office Tracker Scheduler[9] is also a client/server calendaring tool for Windows and Macintosh. Calendars of several people and resource can be viewed side by side. With appropriate privileges, users can schedule for one another and schedule rooms and resources. One version of the server can be accessed and administered via the Web. The system is also able to find open times for people, rooms and resources. Prices (2000-02-23): Single-User $\approx$ 400$, 10 clients $\approx$ 700$.

7.1.6 Other information managers

A few other personal information managers similar to those above were reviewed in 1997 by PC Magazine[10] most of them have evolved since then. They provide calendaring functionality, and have more or less the same "standard" workgroup features: sharing of agendas, control of access rights, "overlaying" of calendars, etc. The names of these products are ACT!, GoldMine, Time and Chaos, OfficeTalk, DayTimer, Commence, etc.

7.2 Discussion of these products

The products described in the previous section intend to help users or groups of users managing their time. Basically, they handle user's agendas, and provide meeting organizers a means of locating free time stretches in the schedules of participants. More advanced functionalities like synchronization with portable devices (Palm, Psion...) and message exchange between participants are also usually available. Some can even take into account the availability of rooms and other necessary resources. Some programs also offer access from a Web browser, or compatibility between different operating systems (Windows, Macintosh, Unix).

There are clear indications that in some contexts these tools are very useful, e.g. Grudin and Palen mention an enquiry conducted with managers and secretaries of a company: "*They collected data and found very large time savings and cost savings due to the online calendar.*" (Grudin and Palen, 1995)

[9]http://www.milum.com/
[10]http://www8.zdnet.com/pcmag/features/infomanagers/_open.htm

However, there are still lots of places where these products are deployed but not used (e.g. Sun Calendar Manager in our Computer Science Department). This fact is confirmed by (Cesta et al., 1998): "*Very often in the past attempts to introduce computerized tools for supporting meeting scheduling failed due to rejection by the user in work environments.*" In the following paragraphs, we'll try to understand why these products failed to attract more users.

7.2.1 Lack of inter-operability

The main shortcoming of these products is that they don't **inter-operate** satisfactorily. As a consequence, they can be used to schedule meetings between people using the same product, for instance inside an organization, but cannot take into account the schedules of people using different products. This fact has been reported by many studies, e.g. (Coleman, 1999):

> *"Heavy schedulers" (those that do a lot of scheduling on a daily basis) report that they have the most difficulty scheduling a meeting with people from both inside and outside the company. It is no surprise, that e-mail and voice mail are the dominant ways to schedule meetings today. However, as we get more sophisticated with the Web and are able to take advantage of its interactive nature we see some solutions in Web-based scheduling.*

This obvious compatibility problem is currently tackled by a working group[11] of the IETF (Internet Engineering Task Force) with representatives from the major software vendors in the field (Microsoft, Netscape, Lotus...). Under the label iCalendar, they are trying to define a set of protocols that would allow inter-operability between their products. Their work includes the development of MIME content types to represent common objects needed for calendaring and group scheduling transactions and access protocols between systems and between clients and servers:

Core Object Specification (iCalendar: RFC 2445) A standard content type for capturing calendar event and to-do information. The content type should be suitable as a MIME message entity that can be transferred over MIME based email systems or the Web.

Calendar Inter-operability Protocol (iTIP: RFC 2446, iMIP: RFC 2447) A standard peer-to-peer protocol for common calendaring and group scheduling transactions. It allows the exchange over the Internet—between different calendaring products—of well-defined messages such as

[11] http://www.ietf.org/html.charters/calsch-charter.html

event-requests, replies to event-requests, cancellation notices for event-requests, free/busy time requests and replies to free/busy time requests.

Calendar Access Protocol (CAP: Internet Draft) A standard access protocol to allow for the management of calendars, events and to-dos over the Internet.

But, at the time of this writing (August 2000), these standards are not finalized and according to one of the working group chairs, the inter-operability of some products, tested in April 2000, is still severely limited[12].

7.2.2 Reliance on public calendars

The second disadvantage of these products is that they rely on the publication of personal calendars for the selection of a date. They have all adopted an approach in which the meeting organizer looks for a possible date by investigating the agendas of all participants. Thus the selection does not really occur in a collaborative way, even if some products like Lotus Notes allow some kind of negotiation (rejecting a proposition, making a counter-proposal). Several inconveniences result from this approach:

confidentiality: even if simple mechanisms are available to protect the privacy of users, by hiding some calendar entries or by masking details of some appointments, some people could find that it reveals too many private informations (Coleman, 1999), this can result in “defensive” actions (blocking out large periods, even if they have no particular appointments) or rejection of the product

completeness: keeping one’s calendar up to date at all times requires a lot of effort and self-discipline; not doing it results in uncertainty and errors, when someone else uses an incorrect calendar to schedule an event; this problem is ranked first in (Coleman, 1999)

constraint: the whole systems works only if everyone complies to the rules; the first consequence is that new people joining a workgroup have to use the same tool, even if they had another personal way to manage their agenda (Grudin and Palen, 1995):

> *How did use become so widespread? When asked directly, some reported peer pressure to keep calendars on-line, some admitted exerting such pressure, others said they did not notice pressure, but might contradict themselves in subtle or not so subtle ways.*

[12]http://www.imc.org/ietf-calendar/mail-archive/msg03499.html

The second consequence is that the whole group has limited access to potentially useful innovations

loss of control: a meeting chair may summon someone he doesn't know, and modify his agenda without consulting him first (Grudin and Palen, 1995):

> *This approach, with no informal prior negotiation of time, can strike outsiders as very blunt when it involves, for example, initiating a one-on-one meeting with someone one does not know and who is not expecting the meeting request.*

The lack of negotiation can also engender more "conflicts through shared resources" (§ 5.4, p. 97) because the owner of the agenda may already be planning an activity for the same stretch of time, without having entered it in the system.

limited applicability: since these systems are based on participant availability, they do not allow the selection of a date in a collaborative fashion, for an event that they will not necessarily attend; there are also additional constraints in meeting scheduling, that are not contained in the participant's agendas, but that may have a strong impact on the decision (anterior and posterior events, project deadlines, etc.)

7.3 Web services

Software that has been presented in the first category is deployed on the user's desktop in the "traditional" way of personal computing. Even if in some cases the servers are able to publish calendar data on the Web, the software is still bought, installed on each computer, and parameterized by the customer.

In this second category, the client-side administrative work is minimized, since calendar information is accessed using a standard Web browser. No installation, no configuration, no backups... The only requirement is to fill a Web form with some personal information, and to choose a password that protects user's privacy. Another advantage is that the calendar is available from any host connected to the Internet. Moreover, these services are usually free of charge, they are offered to increase customer fidelity by the providers who make money from targeted advertisement or sale of other goods.

7.3.1 Online calendars

Netscape (Netcenter) offers a simple calendar to registered users. It can be used to store appointments, and to send a regular email message to a list of people that may be interested in the event. This is the simplest form of calendar services, with no improvement in collaboration, and no support for date selection.

Yahoo's service is a little better, since the organizer of a meeting can see other's calendars. The basic privacy control that is offered by desktop products is also implemented here. However, the calendars are not overlaid. Another limitation is that only Yahoo users can view each other's calendars, thus the usability is very limited.

Besides, all services of this kind suffer from the problems described in § 7.2.2, p. 119, since they rely on calendars, just like the desktop services.

7.3.2 Meeting scheduling services

Products in this category are more interesting than simple online calendars of the previous section, because they are more oriented towards collaboration. These services (Evite[13], TimeDance[14] FireDrop Zaplets[15]) innovate by offering meeting scheduling features, without being calendar oriented. They try to help users planning their parties or meetings, making the communication and organization efforts less time consuming and more pleasant, but they don't provide online calendaring, thus they avoid the pitfalls listed in § 7.2.2, p. 119. Instead, they offer some kinds of message boards, oriented towards meeting planification, that organizers and participants can use in order to collaboratively work on the details of the meeting (time, location, attendance, resources, catering...), by means of email and common Web pages.

These products are similar to Meety (the solution that we propose for our case study), and they appeared at the same time. The main differences can be found in the technologies involved, and in the "orientation" of the product. Given their commercial orientation, the graphical presentation of these services has received much more attention. For instance, the user can choose a layout and decorative effects for his invitation. Our service is much more sober in this respect, but offers a few additional technical features, among others the extension with mobile code. The fact that several similar services appeared at the same time, and their immediate success (TimeDance claimed 21% weekly growth between December 1999 and June 2000) show that our case study corresponds to a real need.

An interesting comparison has been made between the diffusion of these collaborative services and email viruses. Once someone uses the service to schedule a meeting, all the participants are incited to use it too, making it

[13]http://www.evite.com/
[14]http://www.timedance.com/
[15]http://www.zaplets.com/

known to their own acquaintances, and so on. Thus the number of people knowing the service, and using it, can grow very fast. This is quite similar to the behavior of some recent viruses[16] which exploited user naivety to send copies of themselves to all addresses found in the receiver's address book, creating millions of duplicates in just a few hours.

The main differences between **Zaplets** and Meety are: (1) Zaplets are not limited to meeting scheduling but allow invitations without date selection, sharing of information (picture, documents), discussions, polls, and collaborative editing of tables, lists, etc. (2) Zaplets use some advanced features of recent email client programs (Netscape, Outlook), like their ability to display HTML and to execute Javascript, such that the invitation message itself looks like a Web page, which is automatically reloaded every time the message is displayed; it is certainly a nice feature for the users of these programs, but is unacceptable for those who read their messages on text-based clients (3) Zaplets for meeting scheduling are not able to propose more than 5 dates and times to the participants, and this information must be entered textually, whereas Meety doesn't impose a limitation on the number of propositions and reads this information more conveniently and unambiguously with mouse clicks on graphical calendar elements; (4) Zaplets don't have the feature of listing meetings that may conflict with a given time stretch.

The main differences between **Evite** and Meety are: (1) Evite offers three additional services (invitation without collaboration, reminders, polls); (2) Evite provides numerous links to find or buy goods and resources related to meetings and entertainment (mostly in the U.S.); (3) Evite doesn't propose more than 5 dates and times to the participants, and this information must be entered textually; (4) Evite doesn't list meetings that may conflict with a given time stretch, although invitations without collaboration can be displayed on a calendar.

TimeDance was very similar to Meety; more targeted to collaborative date selection than the two other ones, not limiting the number of propositions, it allowed the selection of dates without typing, etc. Curiously, it was discontinued in August 2000, after less than one year of operation. Apparently, the service was successful, but its commercial side was not sufficiently developed to make it profitable.

[16] http://www.cnn.com/2000/TECH/computing/05/04/iloveyou.02/

7.4 Research in distributed constraint satisfaction

This section is about another category of software that also deals with meeting scheduling but from a less commercial perspective. The work presented here is more theoretical, formalizing the problem and its resolution. It explores the consequences of various heuristics that can be applied when several choices are possible, and of various protocols for the negotiation between the participants of a meeting. It also tries to determine what is required for a good interaction between the user and the program. The software developed for these studies is apparently not publicly available, and neither used nor developed actively.

A general definition of scheduling, is provided by (Zweben and Fox, 1994, p. 9):

> *Scheduling is the process of selecting among alternative plans and assigning resources and times to the set of activities in the plan. These assignments must obey a set of rules or constraints that reflect the temporal relationships between activities and the capacity limitations of a set of shared resources. The assignments also affect the optimality of a schedule with respect to criteria such as cost, tardiness, or throughput. In summary, scheduling is an optimization process where limited resources are allocated over time among both parallel and sequential activities.*

This definition encompasses the meeting scheduling problem, as well as numerous other interesting problems that occur in many different contexts: staff scheduling in an organization, machine scheduling in factories (job shop scheduling), transportation scheduling, etc.

When we try to formalize this problem using mathematical notation, we find that it belongs to the larger class of Constraint Satisfaction Problems (CSP). After presenting general solutions for CSP, we will present known techniques to cope with the three characteristics of DEDIS: first distribution and dynamicity which have been most studied, then evolution.

7.4.1 Constraint satisfaction

A Constraint Satisfaction Problem (CSP) is formally defined as:

- n **decision variables** $x_1, \ldots, x_n$;
  for each $j \in \{1, \ldots, n\}$ the value assigned to variable x_j must belong to D_j, a given set called the **domain** of x_j

- m **constraints** $c_1, \ldots, c_m$ which must be satisfied;
  for each $k \in \{1, \ldots, m\}$ the constraint $c_k(x_1, \ldots, x_n)$ is a mathematical relation (a subset S_k of $D_1 \times \ldots \times D_n$); c_k is **satisfied** iff $(x_1, \ldots, x_n) \in S_k$

- an optional **objective function** $f : D_1 \times \ldots \times D_n \to \mathbb{R}$ that provides a ranking of each solution $(x_1, \ldots, x_n)$

This definition is extremely general. The domain of a decision variable can be any possible set, for example the integer numbers, the real numbers, or an enumerated set of colors, of cities, of temporal intervals, etc. The nature of constraints is very general as well, since there is no restriction on the kind of relations imposed between the variables. An equivalent representation of the constraints that is sometimes preferred gives m functions $g_k : D_1 \times \ldots \times D_n \to \{0, 1\}$ where g_k is satisfied iff $g_k(x_1, \ldots, x_n) = 1$.

This generalty of the formalism makes it suitable for modeling a huge number of different decision problems: job scheduling, timetabling, resource allocation, transportation, map coloring, layout of shapes or graphs on 2D surfaces, etc. (Kumar, 1992). In the case of meeting scheduling, one possible model has one variable x_i for each meeting, the domains D_i represent possible start times (Fig. 7.6, p. 125), and the constraints represent the periods when participants are not available (participants can be considered as resources that cannot be allocated to more than one meeting at any given time), as well as other restrictions that organizers want to apply.

Depending on the problem, it may be sufficient to find **one** solution $(x_1, \ldots, x_n)$ such that $x_j \in D_j$ for each $j \in \{1, \ldots, n\}$ and $(x_1, \ldots, x_n) \in S_k$ for each $k \in \{1, \ldots, m\}$. Other cases will require **all** feasible solutions, or only the **optimal** solution, for which the objective function f is maximized or minimized.

Finding these solutions in the potentially huge search space is usually not an easy task. Since the search space is the cartesian product of the domains D_i, its size is the product of their cardinalities (assuming they are finite). For instance, a CSP with $n = 20$ variables, and all the domains containing 2 elements already has 2^{20} elements in its search space. Thus the size of the search space grows exponentially with n, this is noted $\mathcal{O}(e^n)$. For many specific cases it has been shown that the problem is NP-complete, which means that it can be solved in polynomial time by a **non-deterministic** algorithm. Unfortunately, current computers are **deterministic** and solving these problems may require an exploration of the whole search space in the worst case, thus a time complexity in $\mathcal{O}(e^n)$.

Many algorithms and heuristics have been proposed to solve these problems, which depend on the nature of the variables (discrete or continuous) and of the constraints (linear or non-linear). Some well studied special cases are:

i	*possible start* D_i	d_i	*solution* x_i
1	2000-08-28 8:00/16:00, 2000-08-29 8:00/16:00	2h	2000-08-28 14:00
2	2000-08-28 8:00, 2000-08-29 8:00	4h	2000-08-29 8:00
3	2000-08-28 8:00/9:00	1h	2000-08-28 8:00
4	2000-08-29 14:00, 2000-08-30 8:00, 2000-08-31 14:00	4h	2000-08-30 8:00

i is the meeting number, d_i is its duration

$$c_{ij} \equiv (x_i + d_i \leq x_j) \vee (x_i \geq x_j + d_j), \forall i,j \in \{1,2,3,4\}, i \neq j$$

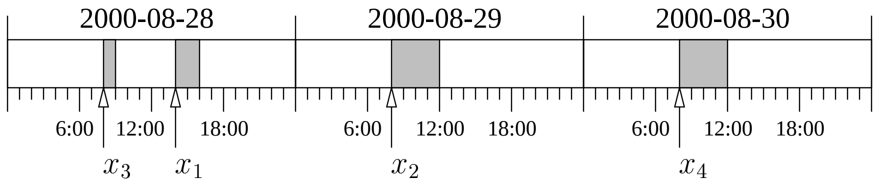


Figure 7.6: Example of a CSP model for meeting scheduling and graphical representation of the solution. Variables represent meetings, domains represent possible start times, constraints ensure that all pairs of meetings are disjoint (cannot overlap).

linear programming: when the variables are continuous:

$$x_j \in \mathbb{R}, x_j \geq 0$$

and the objective function is linear:

$$f(x_1, \ldots, x_n) = c_1 x_1 + \ldots + c_n x_n$$

and the constraints are linear:

$$(x_1, \ldots, x_n) \in S_k \text{ iff } a_{k1} x_1 + \ldots + a_{kn} x_n = b_k$$

the optimum solution can be found using Danzig's **simplex method** (Chvátal, 1983) or **interior points methods** (Hillier and Lieberman, 1990; Nesterov and Nemirovskii, 1994) which have a polynomial time complexity.

nonlinear programming: when the variables are continuous, but the objective function or the constraints are not linear, the problem is NP-complete, but some methods like Numerica can find a global optimum for instances of reasonable size (Van Hentenryck et al., 1997).

integer programming: when the variables take integer values

$$x_j \in \mathbb{Z}$$

the problem is NP-complete and methods for linear programming cannot be applied, even if the objective function and the constraints are linear; integer programming is also called **combinatorial optimisation** indicating that it deals with optimisation problems with an extremely large (combinatorial) increase in the number of possible solutions as the problem size increases, thus making exhaustive search (**generate and test**) completely impractical. The traditional method to solve such problems is **branch and bound** (Garfinkel and Nemhauser, 1972; Held and Karp, 1970): it tries to divide the problem into small sub-problems that can be solved to obtain bounds. These bounds allow the elimination of those sub-problems which have no chance to provide a solution. By examining only a subset of possible solutions, these techniques allow problems of bigger size to be tackled. The performance of this technique is limited by our ability to solve sub-problems and to obtain bounds of good quality (which is easier in the special case of linear constraints).

For other CSPs, which do not fit in one of these categories, several other methods can be applied. These methods can be classified into two categories: the **global** methods that provide a global optimum, and the **local** or **improvement** methods that do not guarantee that the global optimum will be found, but that can be applied when no global method is able to find a solution in a reasonable time. Using local methods is also interesting in many cases either because finding the best solution is not essential, or because the objective function is not clearly defined. This can happen when the real-world problem is difficult to understand and its subtleties are not easy to capture in a mathematical model. This is often the case in scheduling where multiple, conflicting objectives must be achieved. The phenomenon has been described as **scheduling uncertainty** (Berry, 1992; Burke and Prosser, 1994).

heuristic search and constraint programming: the constraint programming (CP) community has developed global search techniques, and their own languages specifically designed to solve CSPs (Van Hentenryck and Saraswat, 1995; Kumar, 1992). There are several tendencies in CP, depending on the underlying theory and the mathematical domains of the variables used to model the problems. The two dominant classes are constraint logic programming (CLP) which is a descendant of logic programming and concurrent constraint programming (cc) with its foundations in the domain of concurrent systems. For both classes, there are specializations depending on the domains of variables: real numbers (R), fractions (Q), integers and finite domains (FD), predicates and finite trees (FT) etc.

One noticeable property of CP systems is that the developer does not need to program the search procedure himself and can focus on his problem and on the best model to solve it. Many algorithms and heuristics to search the solution space efficiently are already available in the CP systems themselves. Moreover, this search can be very efficient because constraints can be used to guide the search process and prune entire regions that contain no solution. An additional benefit is that CP systems provide rich sets of constraints already implemented. With these powerful constraints, the models can be more accurate and easier to develop than with linear or integer programming.

Most of these algorithms perform a depth first search, with more or less the following steps (Nadel, 1989; Kumar, 1992):

1. choose one variable x_j that has not yet received a value
2. choose one possible value from the domain of this variable and assign it to x_j; all the trouble in solving CSPs is that it is not possible to compute which value satisfies all constraints and gives an optimal result; it is sometimes possible to make good guesses using heuristics, but not to make an analytic choice at this stage
3. if it was the last unassigned variable, a solution has been found, either return this solution, or continue searching for other (better) solutions
4. if it was not the last unassigned variable, propagate the constraints by marking as impossible those values in the domains of unassigned variables that are not consistent with existing assignments
5. if constraint propagation has left no possible value in the domain of an unassigned variable x_k, then one of the previous assignments was not allowed by the constraints and must be undone before the search can continue (back tracking)
6. continue search at step 1

At each step, heuristics and techniques can be applied to exploit problem structure and improve search efficiency (Nadel, 1989; Kumar, 1992), frequently mentioned are **variable ordering** heuristics (step 1), **value ordering** heuristics (step 2), **arc consistency** algorithms (step 4), and **back jumping** techniques (step 5).

Some authors have also studied techniques for **Partial Constraint Satisfaction Problems** (Freuder and Wallace, 1992) that can be applied in several difficult cases. It can happen that the set of constraints is so complex that it is not easy to design a model that is not over-constrained, in this case partial constraint satisfaction can find a solution that minimizes the number of constraint violations. Sometimes the CSP must be solved under real-time constraints and obtaining a partial solution in a specific amount of time is better than obtaining a full solution too late.

gradient descent: in all improvement methods the search starts with a non-optimal initial solution and tries to improve it as much as possible. This requires methods to build the initial solution, and operators that compute the neighbors of the current solution. Improvement methods are also called local search methods, because the neighbors are found by changing only small parts of the current solution. Preferably, this does not require a large amount of computation. In the gradient descent method, the neighbor that provides the greatest improvement is chosen at each step, and the search quickly progresses towards better solutions. The problem with this approach is that the objective function may have lots of local minima (or maxima), which are far from being optimal, but where the gradient descent search remains trapped.

simulated annealing: in this second improvement method (Kirkpatrick et al., 1983), the strategy to select the neighbor at each step is different. The goal is to let the search process escape local minima (we assume that the goal is to minimize the value of the objective function), by allowing some "moves" that don't improve the current solution. At each step a neighbor s' of the current solution s is generated randomly. The decision to accept or reject the move from s to s' depends on the change in the objective function $\delta = f(s') - f(s)$. When the new solution is better than the current one ($\delta < 0$), the move is always accepted. Otherwise, the move is accepted with probability $e^{-\frac{\delta}{t}}$ where t is called the temperature. During the search, t is slowly decreased (after a certain number of iterations), thus the probability to accept bad moves becomes smaller and the process eventually settles in a minimum. The name of this method and of the temperature parameter come from the analogy with the annealing process in solid state physics, where a material is cooled down by stages, such that the resulting solid is as stable as possible.

tabu search: this other improvement method (Glover, 1989), also tries to avoid local minima by keeping a list of the best solutions reached during the search, among them the local minima. Returning to these solutions is forbidden, thus the process is able to reach better solutions.

Let s denote an initial solution, and s^* the best solution found so far, which is initially equal to s. The process computes N_s the set of neighbors of s. The best element in N_s, noted s' is selected. If s' is better than s^*, s^* is replaced by s'. The solution s is stored in the list T (the tabu list), s' replaces s and the process continues with the generation of $N_{s'}$.

This time, the best element in $N_{s'}$, noted s'' is chosen from $(N_{s'} - T)$. If s'' is better than s^* then s^* is replaced by s''. The solution s' is stored in T, s'' replaces s' and a new iteration starts.

Because the current solution s is always replaced by a solution $s' \in (N_s - T)$, which may be worse according to the objective function, the search is able to escape local minima. Furthermore, it will not return to an already visited local minimum, as long as it can be stored in the tabu list.

genetic algorithms: another kind of improvement method, where the search does not proceed by improving one single solution, but by evolving a large population of solutions (Holland, 1992). This method is inspired from biological evolution, where individuals with different characteristics are selected by the environment, and only the fittest can reproduce. In the case of CSP, the individuals are solutions, initially scattered at random in the whole search space. The genetic algorithm tries to find an optimal solution by selecting the best individuals of each generation according to the value given by the objective function, and creating new individuals by randomly combining their characteristics.

Whereas the other improvement methods required good techniques to compute the neighbors of the current solution, the performance of genetic algorithms depends on choosing a good encoding of the variables in "chromosomes", and finding the right operators for the combination of individuals (cross-over), such that two individuals with some good and some bad features can produce an individual with only good features in the next generation.

7.4.2 Distributed constraint satisfaction

All the methods for solving CSPs that we have presented so far assume that the search is conducted as a single sequential process, which has at its disposal all the necessary information (variables, domains, constraints and objective function). However, we have seen that meeting scheduling is inherently distributed (§ 5.4, p. 95). In large organizations, the elements of a CSP are also frequently scattered between several departments, with the constraints and variables being controlled in the different units, but with important couplings from the point of view of global resource management (Burke and Prosser, 1994). In (Friha, 1998), the case of the Geneva University Hospital is mentioned, where several clinics (Digestive, Cardiology, Urology, Orthopaedics, etc.) must share the rooms in surgical wings. The clinics are independent to a large extent, but they need to coordinate their use of these shared resources. It would be very inefficient to merge the information systems of all these clinics, just to solve this CSP in a centralized way. In an industrial and competitive context, we cannot imagine that two companies merge their information systems, given the large amount of strategic information they contain. However, they may have compelling economic reasons to co-optimize some inter-dependent parameters of their production processes. In this case, it would be necessary to find values for these parameters that satisfy the goals of both companies, even if the objective functions are not exchanged.

Problems of this kind have been described by (Yokoo et al., 1992; Hamadi, 1999) under the name of **Distributed Constraint Satisfaction Problem** (DCSP). In this context, "mathematical methods" like the Simplex, or Branch and Bound are not applicable, because they need a full representation of the problem in a matrix form. Improvement methods are also handicapped by the difficulty to compute the neighborhoods of a solution when variables and constraints are unavailable to the search process, and by the lack of global objective function. The most successful approach seems to be heuristic search, that can be conducted by several processes asynchronously, communicating to exchange necessary informations. Each process owns a subset of the variables and knows the corresponding domain (a finite set of discrete values) and some related constraints. During the search, values from the domains are assigned to the variables and communicated to other processes which have constraints involving these variables. If an assignment made by process P_1 to variable x_1 contradicts some constraints of process P_2, the latter informs P_1 with a specific message and P_1 must look for another value in the domain of x_1. The main difficulty in this distributed search is to ensure that it terminates and does not keep oscillating because of circular dependencies in the constraints. To solve this issue, a total order relationship must be introduced (Hamadi, 1999, p. 177). Each process systematically sends its assignments to its successors and refers impossibilities to its predecessor and thus the search always terminates, even if the global set of constraints is inconsistent and no solution can be found.

Another interesting research corpus, which lies somewhere between the theoretical work on DCSP and our meeting scheduling service has been described in (Sen and Durfee, 1998):

> *Having evolved out of a tradition of work in Distribued AI (DAI), our approach views meeting scheduling as a distribued task where a separate calendar management process is associated with each person in order to increase reliability and exploit inherent parallelism. Moreover, giving each person his or her own process enhances privacy and permits personal tailoring of preference parameters for scheduling meetings. But because the information about*

available times is distributed among processes that wish to minimize how much information they reveal, arriving at meeting times involves selective, distributed search.

With this approach, the constraint satisfaction part of meeting scheduling (finding a possible time interval for a new meeting i) is delegated by the host of the meeting h_i to his personal **agent**. To perform this task, the agent has access to the agenda of h_i, but not to those of people in A_i the set of attendees, which are kept secret for privacy reasons. An agreement between the agent of h_i and the agents of attendees $a_{ik} \in A_i$ can be reached following the multistage negotiation protocol (Conry et al., 1988). The protocol involves the following steps :

1. on receipt of a meeting to schedule the agent of h_i searches its calendar for n possible intervals
2. then it sends *announce* messages containing the n possible intervals to the agents of attendees
3. an attendee, upon receiveing an *announce* message will return a *bid* message containing n' acceptable intervals; these n' intervals may be a subset of the n proposed intervals, but may also contain new proposals that may accelerate the search process (Sen and Durfee, 1998)
4. the host collects and evaluates these *bids*; if a common possible interval T_i has been found, it is sent in *award* messages to all attendees; otherwise, the bids are rejected and new *announce* messages containing a new selection of n intervals are sent
5. when an attendee receives an *award* message, it checks that its calendar is still free for the given interval T_i and if it is the case books it for meeting i; if T_i is not available any more, the attendee sends a *reject* message

These steps are repeated until a satisfactory schedule is found, or it is recognized that the meeting cannot be scheduled due to the low availability of h_i or of the attendees in A_i. The user may also specify a deadline d_i at which the search process should stop if no solution has been found.

The authors of this work have argued that this negotiation protocol is not extremely sophisticated but has the advantage of being well-defined and "*understandable enough to be embraced by a user*". Additionally, they have thoroughly studied the impact of several parameters like n the number of proposed intervals, as well as different heuristics for the choice of these intervals.

7.4.3 Dynamicity

The other facet of the meeting scheduling problem that is not well treated by mathematical methods is its dynamicity. If we take the simple model of Fig. 7.6, p. 125, where each meeting is a variable whose domain contains possible start times, then constraints limit the possible assignments by forbidding that two meetings overlap if they involve the same resources (people, rooms...). The addition of a new meeting represents an important modification of the problem, since a new variable, with related constraints, must be taken into account. It is very important from a performance point of view that the existing solution can be preserved when the new meeting is incorporated in the system.

With the multistage negotiation protocol described in the previous section, a value for the new variable can usually be found without modifying the existing schedule. However, it may occur that the participants are so busy that they cannot agree on a suitable time. In (Sen and Durfee, 1996; Sen and Durfee, 1995) several heuristics that can be applied during the selection of intervals have been studied. They compare three **search bias** (linear early, linear least dense, hierarchical) according to five criteria (communication cost, iterations, slots searched, meeting hours missed, density profile characteristics). Applying the bias "linear early" implies favouring dates closer to the current time, and results in a calendar with lots of meetings scheduled in the close future and few meetings farther from the current date. With "linear least dense" and "hierarchical" the strategy is to favour the least dense areas in the calendar when scheduling a new meeting and results in an even distribution. Which heuristic is the best depends on the kind of meetings in which the user is involved. It is a common annoyance with CSPs that a heuristic performs very well on some problem instances and very badly on other instances of the same problem. For users with **long meetings**, "linear early" may be more interesting, because it leaves longer free periods of time after the initial (dense) part of the calendar; applying other heuristics leads to a calendar which is more fragmented and where finding room for a long meeting is more difficult. For users involved in **high priority, short notice meetings**, "linear early" is less interesting, because it tends to leave very few opportunities in the close future; on the other hand, "linear least dense", and "hierarchical" tend to leave more opportunities for this kind of meeting.

Rescheduling

As long as new meetings can be incorporated in the existing schedule without cancelling other assignments, the combinatorial explosion that characterizes NP-complete problems is avoided. It is clear that a system that doesn't allow rescheduling is not optimal, since moving a couple of existing meetings could make enough room to incorporate a new one. However, in our service, we have not offered this feature, first because it is probably not necessary for many users, second because we think that the consequences of moving an activity that has already been scheduled are far too complex to be handled automatically without intervention of the meeting chair. What our system does provide is the possibility for the meeting chair to reschedule or cancel a meeting, after having evaluated the potential consequences, for instance by direct communication with the participants. More importantly, the architecture of our system allows extensions that would perform this task in a specific context, and the propagation of the update where necessary (for instance to the calendars of participants).

In a different context, it may be interesting to reschedule a couple of meetings, in order to leave a free interval for the new one. The problem is that shifting a single assignment may not be sufficient, so one is tempted to cancel another one, etc. The amount of work performed to find a solution can become extremely large, and the process may even discover that there is no way to schedule the new meeting, because the overall capacity of the resources is exceeded.

The article (Sen and Durfee, 1996) presents a way to allow rescheduling, while controlling the number of meetings displaced. Their solution is based on the principle that the benefit of adding the new meeting m_i must be greater than the cost of cancelling and rescheduling a meeting m_j. The problem with this approach is to define appropriate cost functions. They show how to take into account the priorities assigned by the users to the meetings, the computational costs, and the probability that m_j cannot be rescheduled after it has been cancelled. The problem is that there are also “external” costs that occur in the real world for each cancellation, and there is problably no practical way to take these costs into account.

Concurrency

The second consequence of dynamicity is that an actor (a user or his agent) can be simultaneously involved in multiple meeting negotiations, with meetings overlapping on some time windows ($D_i \cap D_j \neq \emptyset$). This situation, that we mentioned on p. 97, has been called **Conflict through Shared Resources** in (Sen and Durfee, 1996). These conflicts are inherent to dynamic scheduling and must be taken into account during the negotiation. We can distinguish several approaches:

no reservation: an actor which has found a free time interval I in his calendar may propose it for meetings j and k if $I \subset D_j \cap D_k$. This strategy works as long as I is chosen either for meeting j or for meeting k and not for both. If the two concurrent processes for meetings j and k both select I, then the actor is in trouble, because he cannot attend both meetings. The solution in this case is to keep I assigned to one of the meetings and to start a new negotiation phase for the other one (Sen and Durfee, 1998). Compared to the other approaches, this one is interesting because making no reservations avoids constraining the problem unnecessarily and because no additional data structures are necessary. On the other hand, the cost of rescheduling when a collision occurs can be very high and the request to cancel an assignment makes the protocol more complex.

With Meety this strategy corresponds to a user that is invited to two meetings that may occur at any time during the next week. Because he doesn't want to constrain the problem unnecessarily, and he hasn't got any appointments in his agenda yet, his answers don't exclude any time slots. A collision can occur if (1) there are two organizers, (2) the organizers choose overlapping times for their meetings, (3) the user is not able to update his answer between the two decisions.

blocking proposed intervals: an actor which has found a free time interval $I \subseteq D_j \cap D_k$, and proposes it for meeting j marks it as unavailable and doesn't propose it for meeting k (Sen and Durfee, 1998; Cesta et al., 1998). If the negotiation for meeting j settles on interval I, a conflict has been avoided and the calendar can be updated; if the negotiation for meeting j abandons interval I, the interval becomes available again and can be proposed for meeting k if no other interval has been found inbetween. This strategy avoids conflicts and cancellations but may result in sub-optimal schedules and missed solutions (e.g. if j doesn't use I and k receives another interval I' that could have been used for another meeting...). However, it seems applicable in the case of agents applying the multistage negotiation protocol, where n the number of intervals proposed at each stage may be small enough that only a small part of a user's schedule is blocked at any given time.

For the user of Meety, this strategy corresponds to splitting the next week in two and requesting that the first meeting occurs in the first half and the second meeting in the second half. There are several drawbacks with this strategy (1) when the user is invited to the first meeting, he is probably not aware that he will get a second invitation with the same domain, thus he needs to update his first answer when the second invitation arrives, (2) the odds that the organizers find suitable dates for the meetings are greatly reduced, especially if there are many other participants, (3) proposing only a small number of intervals for each meeting to mimic a multistage negotiation results in more messages exchanged and more work for the participant and the organizer.

We should also note that all attempts to add steps to a protocol without reservations with the hope to avoid cancellations introduce some kind of interval blocking and suffer from the same drawbacks.

minimizing delays: since blocking intervals is not very convenient in Meety, it is better to look for solutions that reduce the risk of collision. This risk can be reduced by a quick update of participants' other answers, as soon as an organizer assigns an interval to a meeting. Thus participant P is involved in meetings j and k, with risks of collisions because $D_j \cap D_k \neq \emptyset$. P's initial answers for j and k both allow interval I because his agenda contains no other activity at this time ($I \subseteq A_{Pj}$ and $I \subseteq A_{Pk}$). When h_j the organizer of j assigns interval I to his meeting, I can be excluded from P's answer for the other meeting very quickly ($A_{Pk} \leftarrow A_{Pk} - I$), thus the fact that P will attend meeting j during interval I can be taken into account by h_k the organizer of the second meeting.

This operation can be performed by the user himself, or by an automatic procedure. It is also possible to implement it when someone's calendar is stored online. In all cases, the probability of a collision is reduced but not completely eliminated. It may still occur that both organizers choose I simultaneously.

With Microsoft Outlook and other calendaring systems that combine the availabilities of all participants at a given instant, the risk of collisions is also minimized, since an organizer may detect an interesting interval and immediately incorporate his meeting into the calendars of all users. However, collisions may still occur if two organizers schedule meetings in the same interval simultaneously (within the time to collect and display all calendars, to make a selection and to update the calendars). Like in the other cases, collisions can be avoided completely only with blocking strategies, for instance by preventing modification of all agendas during the search process.

7.4.4 Evolution and agent parameterization

In order to let personal agents make decisions on behalf of a user, it is necessary to provide them with a lot of up-to-date informations regarding user preferences, habits and environment. Such decisions may include rejecting invitations, guessing at which time the user would prefer the meeting to occur, or estimating that the case is too complex and that the user must enter in the decision process.

Modeling the whole problem as a CSP with constraints and a cost function is impractical and requires too much work. Even if it is feasible, the dynamicity of the problem and the frequent changes in the real-world environment compromise this approach. Thus, more elaborate **knowledge representation techniques** have been applied.

In (Cesta and D'Aloisi, 1999) a system is presented that uses rules and facts to represent three categories of preferences. Temporal preferences are used to assign a degree of availability (high, medium, low, null) to time intervals, according to the type of meeting. Contextual preferences are used to determine the context of a meeting automatically, without intervention of the user; they are based on categories of people, types of meetings, locations of meetings. Autonomy preferences represent user's choices with respect to his privacy and the operations that the agent may perform autonomously.

The context of each meeting is important, since it may change a user's preferences and behavior. It is unlikely that the same time slots are available for professional and leisure activities, or that a meeting invitation coming from a director receives the same attention as those of close colleagues. To take into account changes in context, Cesta et al. propose that the user builds a context hierarchy, where rules can be inherited and refined in each new context (Cesta et al., 1998).

In (Sen et al., 1997), the issue of conflicts in preferences is raised. They apply techniques from voting theory to reach consensus choices for meeting times, that take into account the multiple dimensions (times, locations, peoples, topics, etc.) of a meeting. For each dimension (e.g. weekdays) there are several allowable options (Monday, Tuesday, ...) for which the user can assign a preference value between 0 and 1. The user is also responsible to rate each dimension against each other. Thus their system is able to make a grounded choice, even if no one choice appears to be a clear winner along all dimensions.

Another approach, described in (Maes, 1994) is based on machine learning instead of knowledge representation and rules. The advantage of this approach is that equivalent results may be obtained with less efforts, by avoiding the construction of a model using rules and facts. On the other hand, machine learning needs a large number of examples to acquire the right behaviors, provides answers that are not easy to interprete, thus cannot be trusted easily, and has problems adapting to new contexts and preferences.

In all cases, we see that programming agents that are able to behave similarly to their human owners requires a fair amount of sophistication, both from the programmer and from the end user. Since the goal of Meety was rather to support a large number of users in different contexts, and offer time saving right from the beginning, we didn't try to automate the more complex parts of the decision process. Instead, we chose to design a system whose functionality would be easy to extend and to adapt to the expectations of the users.

Chapter summary

After this overview of existing software, we can summarize the specificities of the Meety service:

- Each user can keep his favourite agenda, since the service is not a calendar.
- Privacy is guatanteed since there is no shared agenda, the reasons to accept or refuse a given date are not communicated.
- It facilitates communication and helps the organizer selecting the best time for his meeting, but doesn't make a decision autonomously.
- Dependences between meetings are not handled by the system (anteriority constraints, additional resources) because the peolple involved in the organization of the meeting are more qualified to understand all constraints; but mobile extensions can be used to automate such tasks (Chap. 11.2).
- It doesn't attempt to cancel and reschedule meetings automatically for the purpose of incorporating a new meeting, thus doesn't suffer from combinatorial explosion.
- It presents all possible solutions but doesn't look for an optimum; the approaches based on priorities or preference levels in answers require more conventions from the users than the current possibility to attach comments to each answer.
- Meeting informations are accessible from any computer connected to the Internet with a standard browser and requires no installation of additional software.
- It can be extended using mobile code, to make the integration with other systems easier, and maintain the coherence of information inside the service automatically, as well as external informations that depend on it.

Chapter 8

Meety implementation

Chapter highlights

- The environment chosen for the implementation: Java is the underlying technology, Voyager is used as a mobile code execution environment, Servlets are used for the Web interface, the relational database MySQL is used for persistence.

- The implementation follows the architecture of Chap. 3: the different components and the different packages that structure the program are described. Their relationships have been carefully chosen such that handling many functional and non-functional aspects is as easy as can be.

- Detailed description of calendaring classes: on top of the couple of calendaring classes present in the Java standard library, we had to implement a few nontrivial classes for the domain of time management. They are described in detail, first to show what kind of programming the problem required, second because we couldn't find equivalent classes or algorithms elsewhere and we think that they could be useful for other projects.

- The conceptual model indicates the size of the project: use-cases guide a concise description of the conceptual model; with 15 tables in the database (and significantly more Java classes), the project is large enough to reveal some real problems, but small enough to be implemented in a few months by a single person. This indicates that we chose the right problem for our case study.

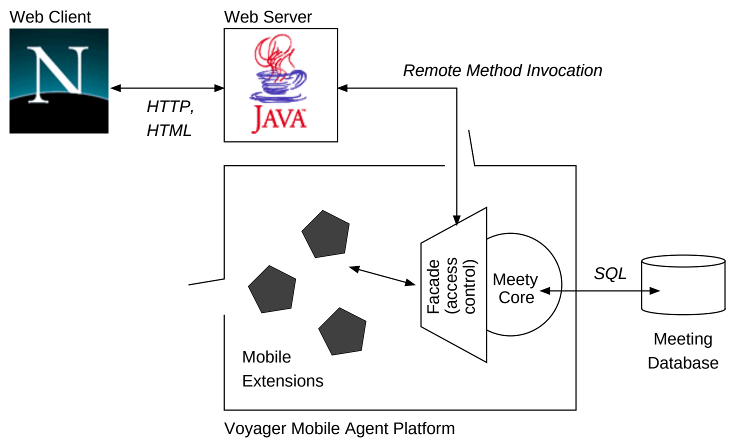


Figure 8.1: The four main components of the architecture and the languages used to connect them. Each component is an independent process, started independently and listening on a predefined communication port.

8.1 Overview

Meety, our Web service for meeting scheduling has been implemented according to the architecture presented in Chap. 3. It is made of a core, which encapsulates a relational database, and of a software interface that can be accessed either through a Web server, or by mobile programs (Fig. 8.1, p. 139).

We have used the mobile code platform Voyager, available as a set of libraries that can be added to the standard Java environment. The main reference for this software (ObjectSpace, Inc., 1999) can be obtained on the Web. We do not describe the product in a dedicated section, but we will present the features that are relevant to our work where necessary. The service is written as a Java application, and the library is used as a middleware, specialized to facilitate communications.

Because communications are mostly handled by the library, and because mobile code guarantees that communication aspects can be treated independently, we were able to concentrate on other aspects while designing the service.

Even if we can delay the choice of a communication protocol, it is essential that the service presents a good interface, that other software components can access. We have identified the following requirements for this interface:

- enable that other components access important informations (informations about meetings and times—the functional aspect of the service)

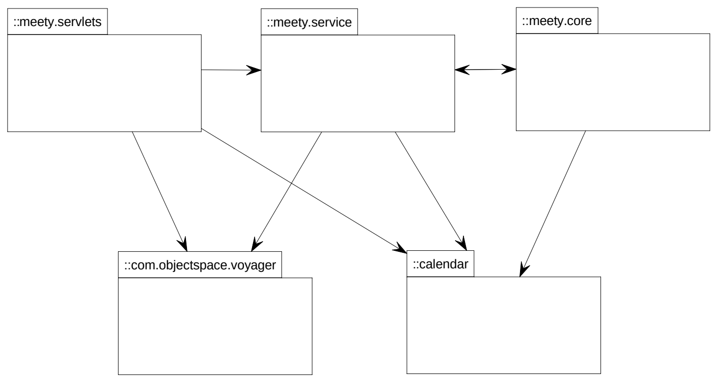


Figure 8.2: The main Java packages, represented by a “folder”, and their dependencies. On the right, “meety.core” represents the inaccessible part of the service (database). The interface of the service is made of “meety.service” that contains specific classes for meeting, and “calendar” that contains more general purpose classes. In the top left corner, “meety.servlets” contains classes for the interaction with the user through the Web. Classes in “com.objectspace.voyager” are used for the communication between the service and its “clients” (“meety.servlets” and mobile codes).

- enable that other components be informed when an event occurs (creation, deletion, modification of an information—the event notification aspect)

- take into account as many other aspects as possible (access control, resource control, etc.) such that the interface doesn't need to be modified, event if the implementation of the service must change.

An overview of the service's Java implementation is given in Fig. 8.2, p. 140. The interface is made of classes in the packages “meety.service”, which contains classes that are specific to meeting scheduling (`Meeting`, `Participant`, `Answer`, etc.); and of package “calendar” that contains general classes for time management (`Month`, `MonthList`, `DayList`, `TemporalDomain`, etc.). All these classes and the relevant algorithms are described in detail in the present chapter. It also describes classes in “meety.core”, which do not belong to the interface, but handle the communication with the database, with the libraries for sending emails, and with an external address management service. Table 8.1, p. 141 summarizes the amount of Java source code that was written to implement the four main parts of the service.

Package	**Nb. of classes**	**Nb. of lines**
“meety.core”	31	4000
“meety.service”	46	3500
“calendar”	53	4500
“meety.servlets”	68	6000
Total	198	18000

Table 8.1: Amount of Java source code for the implementation of the four main parts of Meety.

The interaction with an Internet user running his Web client (browser) is handled by Servlets (Campione et al., 1999). Servlets are Java programs that we have installed on the Web server. These classes are grouped in package “meety.servlets”; their responsibility is to produce HTML pages and formulars displayed by the Web client for the user, and to propagate the actions of the users back to the core of the service, according to the “thin client” model presented in § 3.4.5, p. 65.

For each use-case of Chap. 6 the server executes some specific methods of one or several Servlets. Thus the Servlets play the role of the “interface objects” according to the nomenclature of (Jacobson et al., 1995, § 6.5.1). To implement Meety’s Web interface, we have tried to have a limited number of responsibilities in each Servlet. Hence, there are several of them for each use-case, which roughly correspond to the steps in the use-case or the pages displayed. Which Servlet is invoked depends on the URL, and it receives the state of form controls as parameters. The first action is thus to analyse these inputs, and then to invoke methods of other interface objects. These objects build the answer sent back to the client, which must be a new HTML page. For instance, classes of package “calendar” are responsible to create the table displayed as a small calendar in Fig. 6.5, p. 105.

According to Jacobson’s nomenclature, Servlets also play the role of “control objects”, however, there can be only one instance of each Servlet on the Web server, hence, additional objects are necessary to handle the concurrent interaction with several users. The current state of each concurrent session is stored partly in server-side “session” objects and partly in the information that comes back from the client. Thus, with this structure, there are no explicit “control objects”, the sequence of events for each session is handled jointly by Servlets and session objects. The flow of execution is controlled by providing the right hyperlinks and form actions in the HTML pages sent to the client. When the user chooses a given link or clicks on a form button, he may interact with different Servlets and even different services.

The Servlets represent a second layer of interface for the service (its Web interface) while the main interface is provided by classes in "meety.service". The interaction between Servlets and the main interface is implemented using well-defined instances (e.g. `meety.service.Facade`) and the remote method invocation (RMI) mechanism of Java (§ 4.5, p. 81).

Exactly the same set of instances and methods can be accessed by mobile extensions, in order to carry user-specific tasks. In this case, the method invocations are not performed remotely but locally, since the extensions and the service interface are instantiated in the same Java virtual machine.

8.2 Calendaring

Four standard classes allow the manipulation of dates and times in Java:

`java.util.Date`: This class represents a specific instant in time, with millisecond precision. A long integer (64 bits) is used to store the value, thus can represent dates until the end of year 292272992. When this class is instanciated without parameters, the instance is initialized with the current system time. Instances of this class can be compared with methods `before`, `after` and `equals`.

`java.util.Calendar`: The responsibility of this class is to carry conversions between a `Date` object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. Each instance of `Calendar` stores a `Date` internally. The conversions take into account the complexities of calendaring, like leap years, and are parameterized by the time zone. This class is also able to carry arithmetic operations on the fields, and to compute useful values, such as the day of week.

`java.util.TimeZone`: By choosing between one of the predefined time zones, it is possible to take into account offsets between GMT and local times, as well as daylight savings. The Java run-time system provides a default time zone, which can also be changed programmatically.

`java.text.DateFormat`: The responsibility of this class is to format and parse dates or time in a language-dependent manner, and with various amounts of details.

We have implemented several additional classes to handle abstraction of the calendaring domain. These classes are grouped in package "calendar". They are summarized in Fig. 8.3, p. 143.

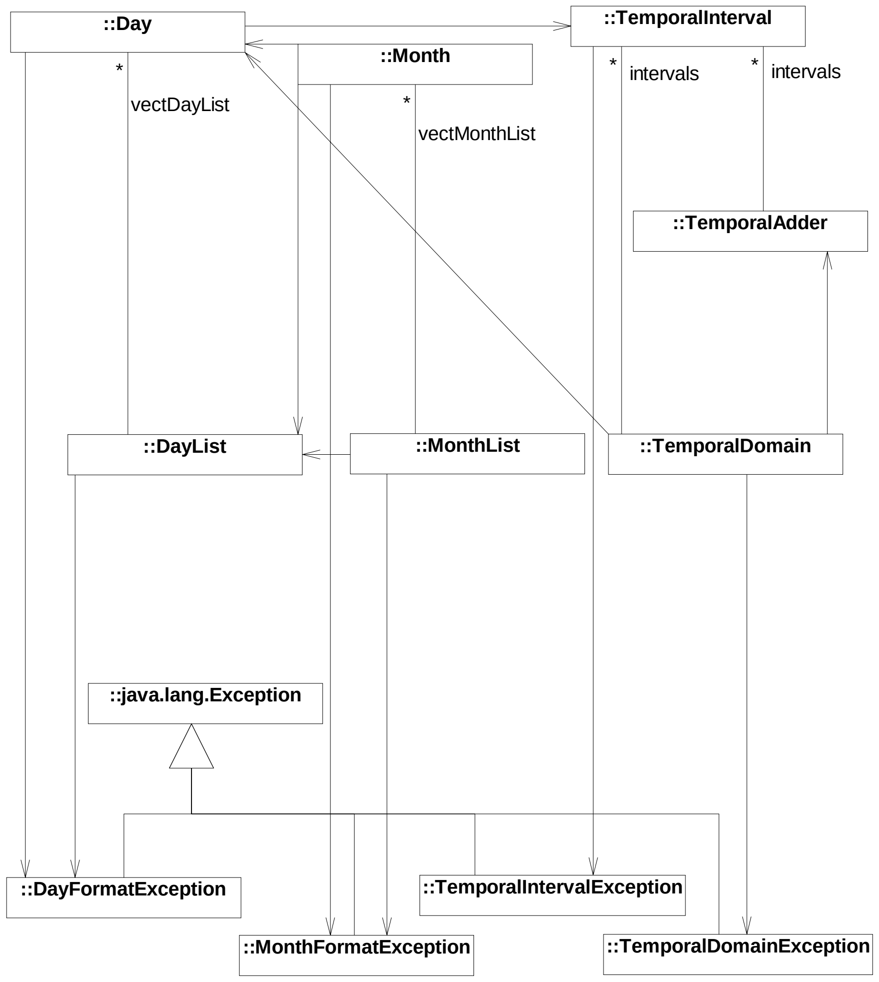


Figure 8.3: UML class diagram representing the main classes in package “calendar”. At the top there are three fundamental abstractions `Day`, `Month` and `TemporalInterval`. Whole groups of these objects can be handled by the lists represented at the middle level (`DayList`, `MonthList` and `TemporalDomain`). At the bottom, there are four new kinds of exceptions that may be thrown when an operation is unable to produce the expected result. More or less all these classes use the four standard classes `Date`, `Calendar`, `TimeZone` and `DateFormat`, thus we omit them for the sake of clarity. Other very common classes like `java.lang.String` or `java.util.Vector` are also omitted.

Package "calendar" contains additional classes for the graphical representation of calendar elements using HTML or Abstract Windowing Toolkit (AWT) objects. We will omit the description of these conceptually simple classes.

8.2.1 TemporalInterval

Instances of class `TemporalInterval` (Fig. 8.4, p. 145) represent a non-empty time period. It is a fundamental abstraction for the domain of time management. The class uses Java's `Date` class to represent the two extreme points "$\min A$" and "$\max A$" of the interval A. Instances of this class are immutable, cloneable and serializable.

Two **constructors** are provided for this class. The first one receives the two instances of `Date` that represent $\min A$ and $\max A$. It throws a `TemporalIntervalException` if $\min A \geq \max A$. The other constructor takes a `String` and parses it. It expects two long integers, separated by a space and surrounded by square brackets. The two integers are converted to instances of `Date`, and an exception is thrown if the parsing fails, or if the values are not ordered as expected.

One important responsibility of this class is to perform **comparisons** of intervals. There are thirteen possible temporal relationships between intervals, that were identified and named in (Allen, 1983). We have implemented thirteen corresponding methods, summarized in Fig. 8.5, p. 146, returning a boolean value, which are very useful for all subsequent operations. For each method, the answer is obtained with a minimal number of comparisons.

A second version of the `contains` method takes a `Date` as parameter and returns true if the `Date` is inside the interval (the interval is closed, thus the answer is "true" for both end points).

We have implemented 9 additional comparison functions. Although they are all combinations of the basic 13 relationships, they are exploited for the addition and subtraction of intervals, and can be performed more efficiently and with fewer tests than a combination of basic comparisons. Their mathematical definition is given in Fig. 8.6, p. 146.

Three methods return a new `TemporalInterval` that can be easily computed from another one: `everythingAfter` that returns an interval going from the end of the current one to the highest possible `Date`, `everythingBefore`, that returns an interval going from the earliest possible `Date` to the beginning of the current interval, `extendedToEndOf` that returns a new interval beginning like the current one and ending like the given one. All these methods throw a `TemporalIntervalException` if the extremities of the new interval are not suitably ordered. We have also implemented a static method `everything` that returns the largest possible interval.

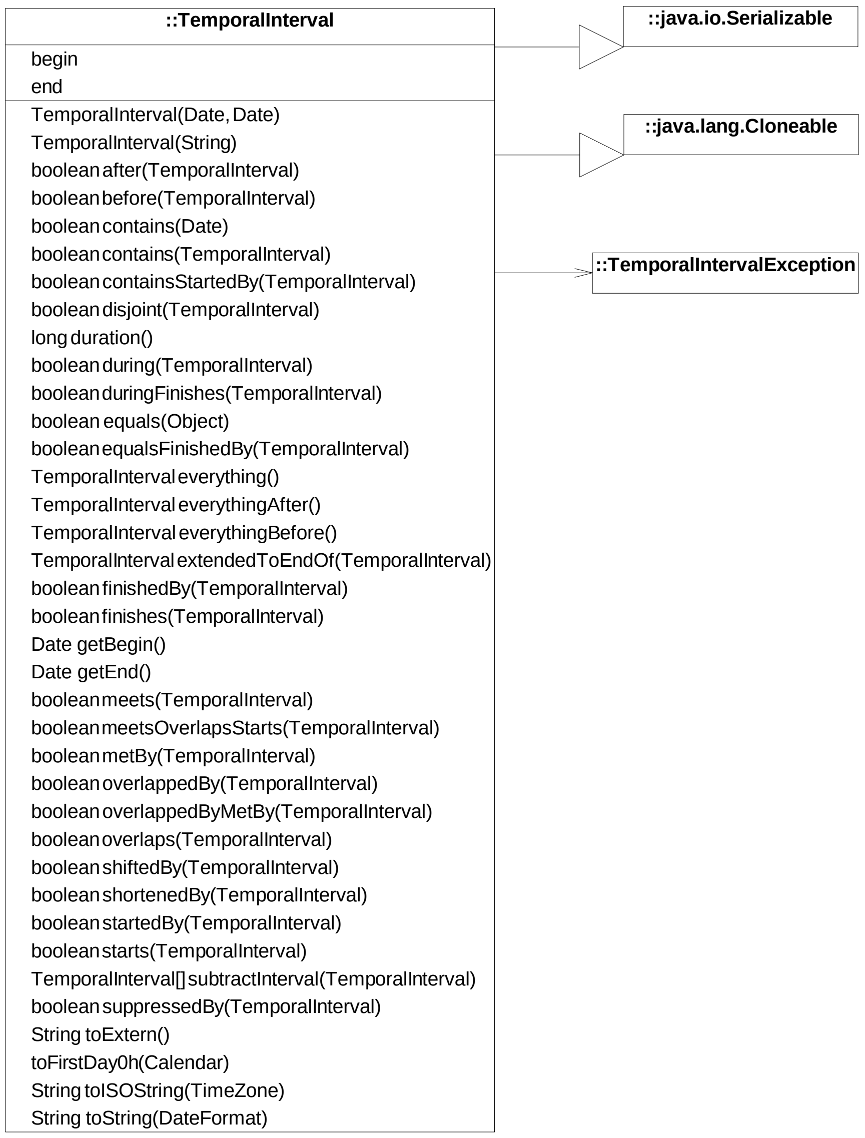

::TemporalInterval
begin
end
TemporalInterval(Date, Date)
TemporalInterval(String)
boolean after(TemporalInterval)
boolean before(TemporalInterval)
boolean contains(Date)
boolean contains(TemporalInterval)
boolean containsStartedBy(TemporalInterval)
boolean disjoint(TemporalInterval)
long duration()
boolean during(TemporalInterval)
boolean duringFinishes(TemporalInterval)
boolean equals(Object)
boolean equalsFinishedBy(TemporalInterval)
TemporalInterval everything()
TemporalInterval everythingAfter()
TemporalInterval everythingBefore()
TemporalInterval extendedToEndOf(TemporalInterval)
boolean finishedBy(TemporalInterval)
boolean finishes(TemporalInterval)
Date getBegin()
Date getEnd()
boolean meets(TemporalInterval)
boolean meetsOverlapsStarts(TemporalInterval)
boolean metBy(TemporalInterval)
boolean overlappedBy(TemporalInterval)
boolean overlappedByMetBy(TemporalInterval)
boolean overlaps(TemporalInterval)
boolean shiftedBy(TemporalInterval)
boolean shortenedBy(TemporalInterval)
boolean startedBy(TemporalInterval)
boolean starts(TemporalInterval)
TemporalInterval[] subtractInterval(TemporalInterval)
boolean suppressedBy(TemporalInterval)
String toExtern()
toFirstDay0h(Calendar)
String toISOString(TimeZone)
String toString(DateFormat)
::java.io.Serializable
::java.lang.Cloneable
::TemporalIntervalException

Figure 8.4: `TemporalInterval`, its attributes, methods and main relationships to other classes.

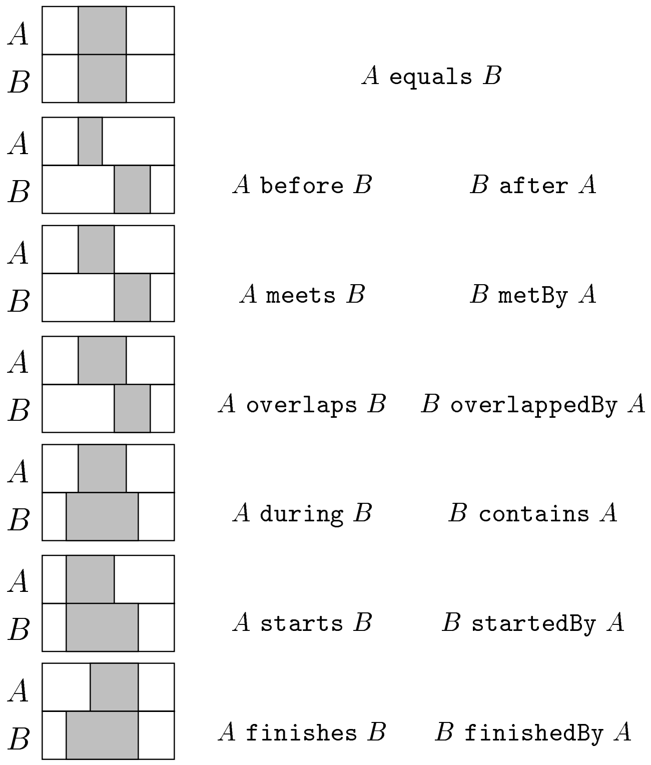


Figure 8.5: The thirteen relationships between two temporal intervals. The gray areas represents the intervals, the time runs along the abscissa.

Relationship	Definition
`containsStartedBy`	A `contains` $B \vee A$ `startedBy` B
`disjoint`	A `before` $B \vee A$ `meets` $B \vee A$ `metBy` $B \vee A$ `after` B
`duringFinishes`	A `during` $B \vee A$ `finishes` B
`equalsFinishedBy`	A `equals` $B \vee A$ `finishedBy` B
`meetsOverlapsStarts`	A `meets` $B \vee A$ `overlaps` $B \vee A$ `starts` B
`overlappedByMetBy`	A `overlappedBy` $B \vee A$ `metBy` B
`shiftedBy`	A `startedBy` $B \vee A$ `overlappedBy` B
`shortenedBy`	A `overlaps` $B \vee A$ `finishedBy` B
`suppressedBy`	A `starts` $B \vee A$ `during` $B \vee A$ `finishes` $B \vee A$ `equals` B

Figure 8.6: Additional relationships and their definitions in terms of basic relationships.

The method `subtractInterval` performs the subtraction of another interval B from the current one A. Depending on the case, it may return 0, 1 or 2 intervals. It has been implemented by returning a variable-size array with the right number of instances of `TemporalInterval`. The method must distinguish five cases:

1. A `suppressedBy` B: the resulting array has 0 element
2. A `disjoint` B: the resulting array has 1 element (A)
3. A `shortenedBy` B: the resulting array has 1 element ($[\min A, \min B]$)
4. A `shiftedBy` B: the resulting array has 1 element ($[\max B, \max A]$)
5. A `contains` B: the resulting array has 2 elements ($[\min A, \min B]$ and $[\max B, \max A]$)

We provide four methods to **read values** related to the interval: `getBegin`, `getEnd` that return a `Date`; `duration` that returns a long integer that represents the duration in milliseconds; `toFirstDay0h` that takes the beginning of the interval A, inserts it into the given instance of `Calendar` and clears the hour, minute, second and millisecond fields, such that the `Calendar` is positioned at the beginning of the day that contains $\min A$. This last operation is implicitly dependent of the current time zone of the `Calendar`.

There are also three methods that return a new `String`: `toExtern` returns the integers corresponding to $\min A$ and $\max A$ within square brackets (what the second constructor expects); `toISOString` returns the hours and minutes of $\min A$ and $\max A$ with the separator recommended by the ISO standard (e.g. 08:00/15:00), and is parameterized by the time zone; `toString` applies the given `DateFormat`, with its own time zone to both end points and returns them between square brackets and separated by a comma.

8.2.2 TemporalDomain

A temporal domain D can be represented by an ordered list of disjoint temporal intervals

$$D = \bigcup_{j=1}^{n} I_j \text{ where } I_k \texttt{ before } I_{k+1}, \forall k \in \{1, \ldots, n-1\}$$

In our design, they are represented by instances of `TemporalDomain` (Fig. 8.7, p. 148). These instances are cloneable and serializable, but unlike the instances of `TemporalInterval` they can be changed by some methods.

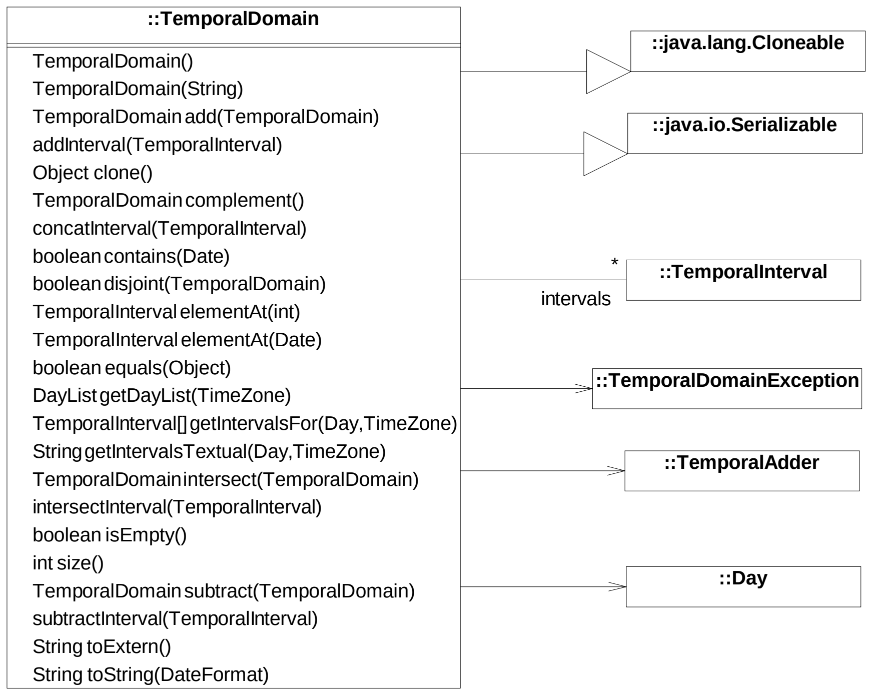

::TemporalDomain
TemporalDomain()
TemporalDomain(String)
TemporalDomain add(TemporalDomain)
addInterval(TemporalInterval)
Object clone()
TemporalDomain complement()
concatInterval(TemporalInterval)
boolean contains(Date)
boolean disjoint(TemporalDomain)
TemporalInterval elementAt(int)
TemporalInterval elementAt(Date)
boolean equals(Object)
DayList getDayList(TimeZone)
TemporalInterval[] getIntervalsFor(Day,TimeZone)
String getIntervalsTextual(Day,TimeZone)
TemporalDomain intersect(TemporalDomain)
intersectInterval(TemporalInterval)
boolean isEmpty()
int size()
TemporalDomain subtract(TemporalDomain)
subtractInterval(TemporalInterval)
String toExtern()
String toString(DateFormat)
::java.lang.Cloneable
::java.io.Serializable
*
::TemporalInterval
intervals
::TemporalDomainException
::TemporalAdder
::Day

Figure 8.7: `TemporalDomain`, its attributes, methods and main relationships to other classes.

There are two **constructors** for this class, one that takes no parameter and builds an empty temporal domain, and one that takes the description of a temporal domain encoded in a `String`. This encoding is made of a prefix, of the number of intervals between parentheses, and of the external representation of the intervals themselves. This constructor may throw a new `TemporalDomainException` if the encoding rules are not respected.

The method `clone` can be called to obtain a copy of a domain. The new instance contains its own list of intervals, but the intervals themselves don't need to be duplicated, since there is no way to modify them.

Four methods **modify** the contents of the temporal domain D. All modifications take a `TemporalInterval` I as parameter. The simplest one is `concatInterval` which appends I at the end of D, without checking that $I \cap D = \emptyset$, thus this operation must be used with care, or the new state of D may be inconsistent with our definition. The method `addInterval` must be used to take into account the possible intersections between D and I. This rather complex operation is delegated to an instance of `TemporalAdder` (§ 8.2.3, p. 150). The method `subtractInterval` is performed by replacing each I_j of D with $I'_j = I_j \setminus I$, where the subtraction is performed by method `subtractInterval` of class `TemporalInterval` (p. 147). The temporal complexity of this operation is $\mathcal{O}(n)$ where n is the number of intervals in D. The method `intersectInterval` is performed by subtracting the two intervals $[-\infty, \min I]$ and $[\max I, \infty]$ from D.

Four methods **build** a new temporal domain D' from the current one D: (1) With the method `complement`, $D' = [-\infty, \infty] \setminus D$, the new domain contains all intervals that are not in D. The complement can be computed in a single "pass" by examining the beginning and the end of each interval in D, thus its temporal complexity is $\mathcal{O}(n)$ where n is the number of intervals in D. (2) With the method `add`, which takes an other temporal domain O as parameter, $D' = D \cup O$. This operation is implemented by making a copy of D using the `clone` method and calling `addInterval` for each interval in O. The resulting temporal complexity is $\mathcal{O}(n \cdot m)$ where n is the number of intervals in D and m is the number of intervals in O. (3) With the method `subtract`, $D' = D \setminus O$. The implementation of this method is similar to the previous one. (4) With the method `intersect`, $D' = D \cap O$. The result is obtained in two phases, first using `complement` to compute $E = [-\infty, \infty] \setminus O$, then using `subtract` to compute $D' = D \setminus E$. It is certainly possible to optimize `add`, `subtract` and `intersect`, but we have preferred to rely on our previous implementations of `addInterval` and `subtractInterval` that are already quite elaborate, since the performances were sufficient for our project.

Four boolean methods **check** certain conditions: `contains` returns true if the given date is within one of the intervals of D; `disjoint` checks whether the intersection of two domains is empty; `equals` verifies that the intervals in two domains are the same; `isEmpty` is true if the domain contains no interval.

The current contents of the domain can also be **accessed** in several ways. The number of intervals n is returned by method `size`, and the ith interval is returned by method `elementAt`. A second version of `elementAt` returns the interval that contains a given `Date`, or null if no such interval can be found. The method `getDayList` returns the list of days "touched" by the domain (§ 8.2.4, p. 154); the temporal complexity of this operation is $\mathcal{O}(n \cdot m)$ where n is the number of intervals in D and m is the number of days, since `subtractInterval` is applied for each day found in the domain. With `getIntervalsFor` all intervals for a given day are returned. First the interval I that corresponds to the day is computed (§ 8.2.4, p. 154), then the intersection of D and I is computed with `intersectInterval`, finally an array with the remaining intervals is returned. A similar method is `getIntervalsTextual`,

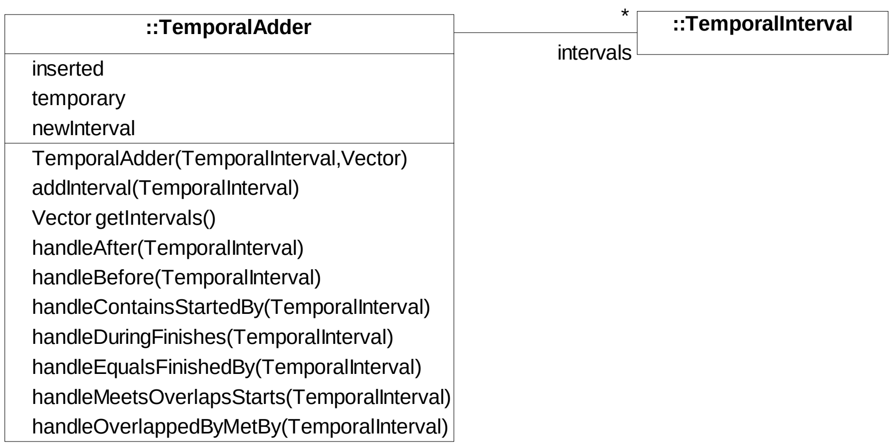

::TemporalAdder
inserted
temporary
newInterval
TemporalAdder(TemporalInterval,Vector)
addInterval(TemporalInterval)
Vector getIntervals()
handleAfter(TemporalInterval)
handleBefore(TemporalInterval)
handleContainsStartedBy(TemporalInterval)
handleDuringFinishes(TemporalInterval)
handleEqualsFinishedBy(TemporalInterval)
handleMeetsOverlapsStarts(TemporalInterval)
handleOverlappedByMetBy(TemporalInterval)
*
intervals
::TemporalInterval

Figure 8.8: `TemporalAdder`, its attributes, methods and main relationships to other classes.

which calls `getIntervalsFor` and then returns a line describing the start and end times of each interval found.

The two last methods provide a readable **representation** of the domain; `toExtern` provides the representation expected by the second constructor (a prefix followed by n between parentheses followed by the intervals I_k). The method `toString` returns the same kind of description but applies the provided `DateFormat` to the end points of each interval (instead of the long integers that are generated by the method `toExtern` of class `TemporalInterval`).

8.2.3 TemporalAdder

The class `TemporalAdder` (Fig. 8.8, p. 150) is responsible for the addition of a new interval J to an existing domain $D = \bigcup_{k=1}^{n} I_k$. This addition is not easy because J may intersect any of the I_k and all intervals must be disjoint and ordered in the resulting domain D'. On the other hand, knowing that the I_k are disjoint and sorted makes many other operations easier.

The parameters of the **constructor** are the `TemporalInterval` J and a `Vector` containing the I_k. This function initializes attributes of the instance and calls `addInterval` for each I_k. When the constructor returns, the insertion is finished and method `getIntervals` may be called to retrieve the elements of D'.

For each I_k, `addInterval` is invoked by the constructor. This method

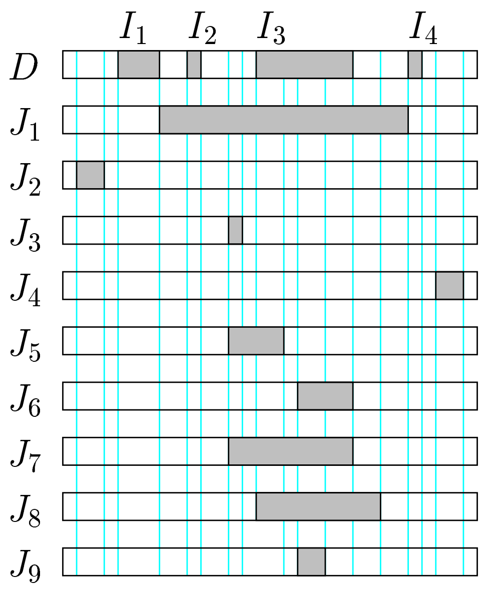


Figure 8.9: Illustration for the insertion of an interval into a domain. The domain that must be updated is D. The J_k represent different possible configurations but the algorithm considers only one J at a time.

compares J and I_k and passes I_k to one of the seven `handle...` methods, according to the result of the comparison. The different cases referenced are illustrated in Fig. 8.9, p. 151. Before we can explain how the intervals are handled, we need to describe the attributes of the class more precisely:

- `newInterval` is a reference to the `TemporalInterval` J
- `intervals` is an initially empty list that will be filled when the I_k in domain D are examined
- `inserted` is a boolean value that is initially false and that is set to true when J is "encountered" by the algorithm; it is used for instance to handle the special case of J_4 which is not encountred when the algorithm loops on the I_k
- `temporary` is a pointer to a `TemporalInterval` that cannot be inserted in `intervals` because its end is not yet determined; for instance in the case of J_1, it holds $[\min I_1, \max J_1]$ while the algorithm examinates I_2 and I_3.

The seven `handle...` methods receive the “current” interval I_k:

1.
```
handleBefore is called when $I_k$ before $J$ (for instance $I_1$ and $J_3$):
    - append $I_k$ to intervals
```

2.
```
handleAfter is called when $I_k$ after $J$:
    - If inserted is false (for instance $I_1$ and $J_2$)
        - append $J$ to intervals
        - append $I_k$ to intervals
        - inserted $\leftarrow$ true
    - else (inserted is true)
        - If temporary is null (for instance $I_2$ and $J_2$)
            - append $I_k$ to intervals
        - else (temporary is not null, for instance $I_4$ and $J_8$)
            - append temporary to intervals
            - append $I_k$ to intervals
            - temporary $\leftarrow$ null
```

3.
```
handleMeetsOverlapsStarts is called when $I_k$ meets, overlaps or starts $J$ (for instance $I_1$ and $J_1$):
    - inserted $\leftarrow$ true
    - temporary $\leftarrow$ $[\min I_k, \max J]$
```

4.
```
handleDuringFinishes is called when $I_k$ during or finishes $J$
    - If inserted is false (for instance $I_3$ and $J_7$)
        - inserted $\leftarrow$ true
        - temporary $\leftarrow$ $J$
    - else (inserted is true, for instance $I_3$ and $J_1$)
        - nothing to do
```

5.
```
handleEqualsfinishedBy is called when $I_k$ equals or finishedBy $J$ (for instance $I_3$ and $J_6$):
    - inserted $\leftarrow$ true
    - temporary $\leftarrow$ $I_k$
```

6.
```
handleContainsStartedBy is called when $I_k$ contains or
startedBy $J$ (for instance $I_3$ and $J_9$):
    - inserted $\leftarrow$ true
    - append $I_k$ to intervals
```

7.
```
handleOverlappedByMetBy is called when $I_k$ overlappedBy or
metBy $J$
    - If inserted is false (for instance $I_3$ and $J_5$)
        - inserted $\leftarrow$ true
        - append $[\min J, \max I_k]$ to intervals
    - else (inserted is true, for instance $I_4$ and $J_1$)
        - append $[\min$ temporary$, \max I_k]$ to intervals
        - temporary $\leftarrow$ null
```

After looping on all I_k, D' is not always fully computed. The last part of the algorithm ensures that J has actually been encountered and that the pointer `temporary` is null:

8.
```
    - If inserted is false (for instance $J_4$)
        - inserted $\leftarrow$ true
        - append $J$ to intervals
    - else (inserted is true)
        - If temporary is not null
            - append temporary to intervals
            - temporary $\leftarrow$ null
        - else (temporary is null)
            - nothing to do
```

The following properties must be verified to ensure that our algorithm is correct:

- **The algorithm terminates.** Since each I_k in D is examined only once, the algorithm terminates after n steps. Furthermore, the number of comparisons and operations for each step can be bounded by a constant value, thus the temporal complexity of the algorithm is $\mathcal{O}(n)$.
- **The 13 possible cases are handled.** The seven `handle...` functions cover all possible relationships between I_k and J. In each case, the function that must be executed is uniquely determined by the tests carried in `addInterval`.

- **The intervals in D' are disjoint and sorted.** This property is guaranteed because (a) The intervals in D are disjoint and sorted. (b) The intervals I_k are visited in sequence. (c) All functions that append an interval to `intervals` (1,2,6,7) either make sure that it is disjoint from others or perform the necessary combination of intervals. (d) When two intervals are appended, the order is preserved (2).

- **J is always present in D'.** If J is disjoint from all I_k it is inserted by the first branch of `handleAfter` (2) or by the final step (8). If J is contained in one of the I_k, it doesn't need to be inserted (5,6). If J intersects one of the I_k, the interval is widened (3,4,7).

- **All I_k are always present in D'.** In functions (1,2,6) I_k is appended directly. In (4) I_k is contained in J and doesn't need to be inserted. In (7) the interval is widened to include I_k. In (3,5) I_k or a widened interval containing it is copied to `temporary`, which is always inserted in `intervals` (2,7,8).

8.2.4 Day

Instances of class `Day` (Fig. 8.10, p. 155) represent one day. Once an instance is created, its attributes cannot be changed. The class is cloneable, as well as serializable.

There are three **constructors** for this class, one that takes three integers values for year, month and day, with the particularity that the month must take a value between 0 (January) and 11 (December). The second constructor takes a `String`, structured according to the ISO standard representation (International Organization for Standardization, 1988). A date is represented by "*year–month–day*", where *year* is unabridged, *month* is a two digit value between 01 (January) and 12 (December), and *day* is a two digit value between 01 and the number of days in the month. The third constructor of the class is able to initialize the fields according to the values in the provided `Calendar`. All these constructors pass three integers to `initialize`, which checks that the values are within acceptable ranges before storing them in the attributes of the instance. A `DayFormatException` is thrown if the instance cannot be initialized properly.

Important methods are `before`, `after` and `equals` that allow the **comparison** of two instances by comparing their fields. This efficient comparison is made possible by the "normalization" step in method `initialize`.

Methods `dayOfWeek`, `isBusiness`, `isWeekend` are useful to **select** certain days in a set, or to line them in colums on user interfaces. The computations

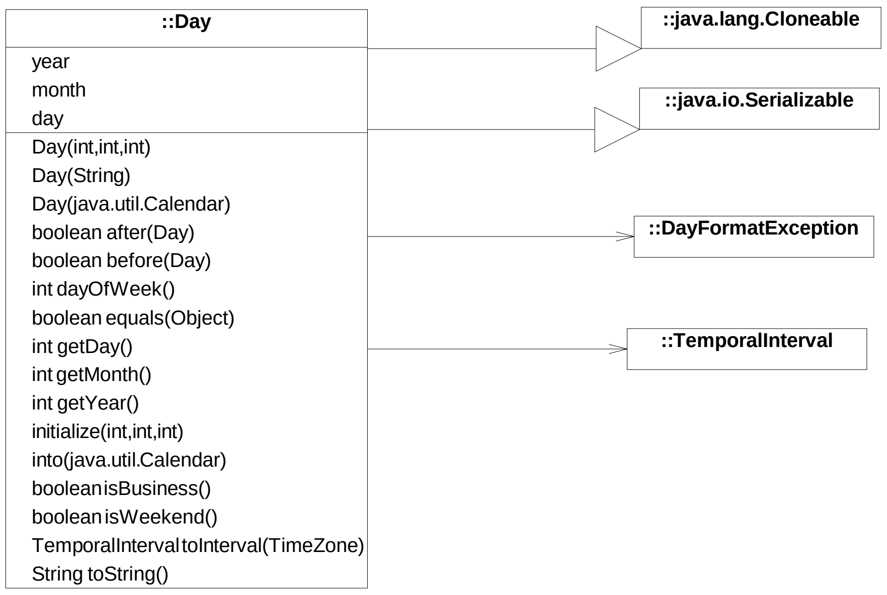


Figure 8.10: `Day`, its attributes, methods and main relationships to other classes.

are delegated to instances of `Calendar` that are created when the methods are invoked.

The contents of `Day` instances can be **accessed** in several ways: `getDay`, `getMonth`, `getYear` return the corresponding fields, `into` writes the fields into the provided instance of `Calendar`, `toString` represents the day according to the ISO standard.

The other way to access the contents of a `Day` is to use method `toInterval`. This method returns a new `TemporalInterval` that represents the day, according to the given `TimeZone`. Fig. 8.11, p. 156 illustrates why it is necessary to take the time zone into account.

8.2.5 DayList

Instances of class `DayList` (Fig. 8.12, p. 157) contain references to an arbitrary number of days. They can be serialized, cloned and modified during their existence. The days are stored in chronological order, and the same day can be referenced several times.

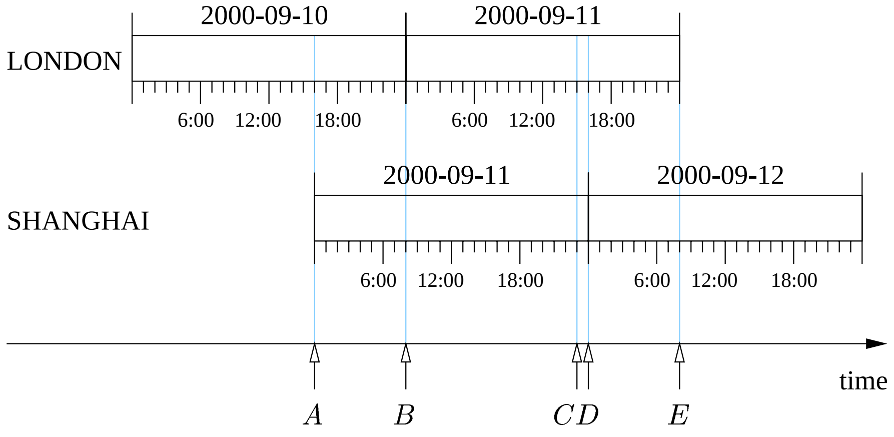


Figure 8.11: At the point in time C, the local time in London is 2000-09-11 15:00 GMT, and the local time in Shanghai (GMT+8) is 2000-09-11 23:00 CTT. This means that the "day 2000-09-11" has started **15 hours** earlier for someone in London (at point B) and it will last for 9 more hours (to point E). But for someone in Shanghai, the same "day 2000-09-11" has started **23 hours** earlier at point A (it was 2000-09-10 16:00 GMT) and will last just one more hour to point D, (it will be 2000-09-11 16:00 GMT). This example illustrates that the temporal interval that corresponds to a day depends on the time zone.

The class has two **constructors**. The first one initializes an empty list. The second one receives a `String` that represents the days of the list in ISO format, separated by a vertical delimiter (|). This external representation is produced by method `toString`.

New instances can also be produced by the `clone` method, which builds a new `DayList` with its own `Vector`. The new list can safely refer to the same instances of `Day` since these instances cannot be modified.

Two instances can be **compared** using `equals`. This method returns true if all days in both lists are equal.

The basic **list manipulation** methods are provided: `addElement` inserts a `Day` into the list, according to the chronological order, but without checking whether it's already present, `size` returns the number of elements in the list, `elementAt` returns the ith element, `contains` checks if a `Day` is present in the list, `remove` receives a `Day` and deletes the first corresponding instance from the list.

Subsets of a `DayList` L are returned by five methods: `businessDays` re-

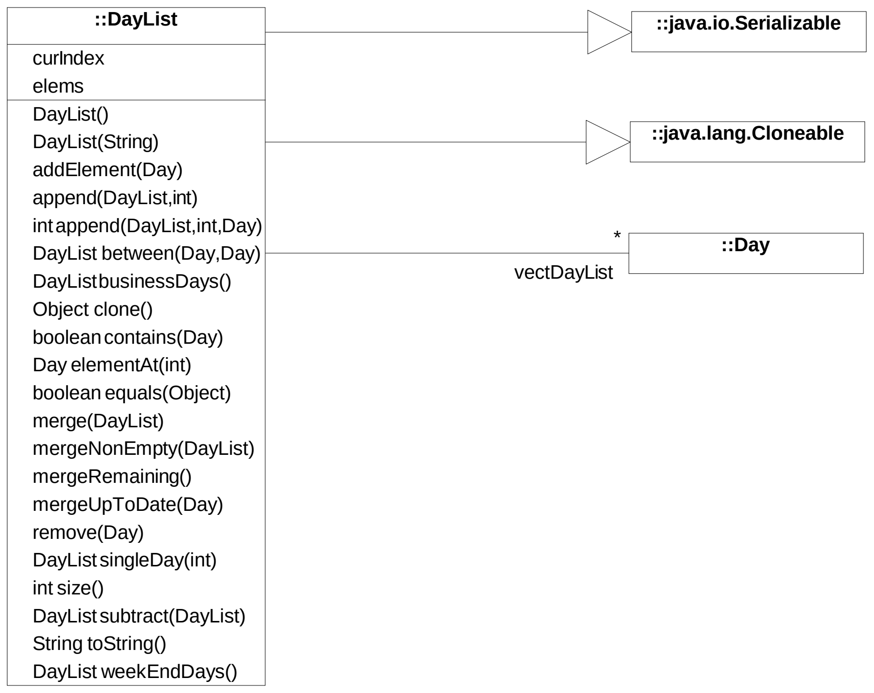


Figure 8.12: `DayList`, its attributes, methods and main relationships to other classes.

turns a new `DayList` that contains only the elements of L that correspond to Monday, Tuesday, Wednesday, Thursday or Friday in the Gregorian calendar. Similarly, `weekEndDays` returns a new `DayList` with only Saturdays and Sundays. With `singleDay`, instances corresponding to one single day are returned.

Another method of this category is `between` that returns a new `DayList` containing only the element of L that are between two given days (inclusively). Finally, `subtract` returns a new `DayList` that contains only the elements of L that are not present in the provided list. Elements of both lists are examined only once, thus the temporal complexity of this method is $\mathcal{O}(n)$ where n is the number of elements in L, as for all other methods returning a subset. The two versions of `append` are used by `subtract` to copy elements from a given index (the integer parameter) up to an optional limit (the `Day` parameter).

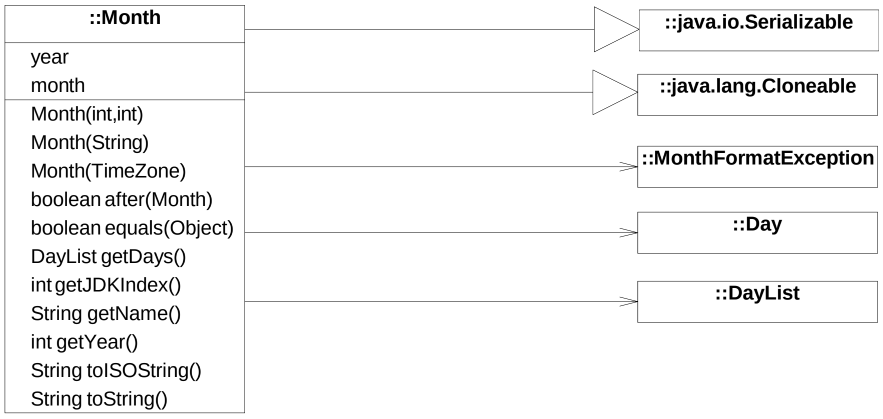


Figure 8.13: `Month`, its attributes, methods and main relationships to other classes.

Inserting the contents of another `DayList` into an existing one is performed by `merge`. This method preserves the chronological order and keeps all elements, even if there are duplicates. By using the helper methods `merge...` and the attributes `curIndex` and elems, this methods is able to merge both lists in $\mathcal{O}(n)$ steps.

8.2.6 Month

Instances of class `Month` (Fig. 8.13, p. 158) represent a month in a similar way that instances of `Day` represent days. These instances are cloneable and serializable; they cannot be modified after their creation.

Three **constructors** can be used to create intances. The first one takes a year and a month (between 0 and 11) and stores them in the attributes. If the month is not in the expected range, a new `MonthFormatException` is thrown. The second constructor takes an ISO representation of the month with a fully represented year and two digits for the month, between 01 and 12. If the `String` is not well-formed, a `MonthFormatException` is thrown. The third constructor initializes the new instance to the current month, which is computed using the system time and the given `TimeZone`.

Months can be **compared** using methods `after` and `equals`.

The contents of an instance can be **accessed** using methods `getYear` and `getJDKIndex`. The latter returns the integer between 0 and 11 that represents the month.

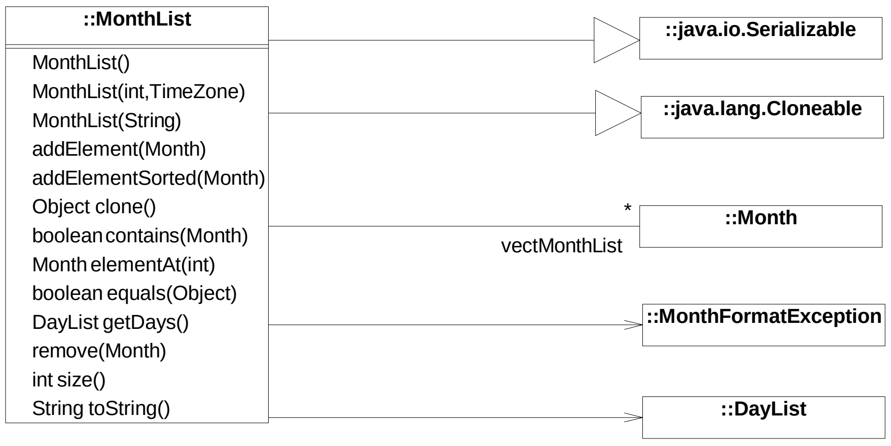


Figure 8.14: `MonthList`, its attributes, methods and main relationships to other classes.

A month can be **represented** in ISO format using `toISOString`, a textual form comprising the month name can be obtained with `toString`. The name only can also be obtained with `getName`.

Method `getDays` creates a new `DayList` and initializes it with all days of the month.

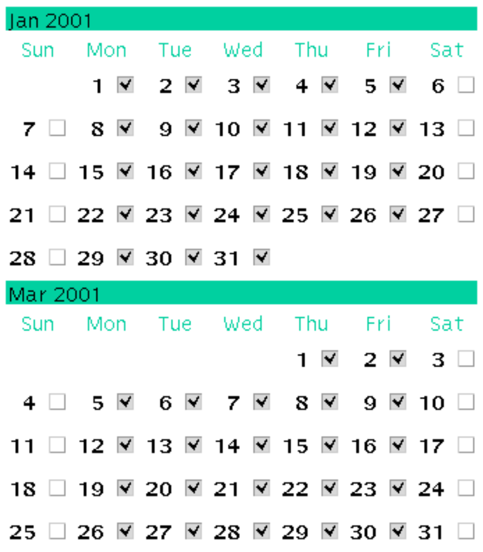
8.2.7 MonthList

Instances of class `MonthList` (Fig. 8.14, p. 159) represent a list of months, in a similar way that instances of `DayList` represent a list of days. Instances of this class can be serialized, cloned and modified. The months can be stored in chronological order, and may be present several times in the list.

The first **constructor** takes no parameter and returns an empty list. The second one receives an integer n and a `TimeZone`. It initializes the list with n consecutive months, starting with the current one (for the given time zone). The third constructor takes a `String` where months to insert in the list are represented in ISO format, separated by vertical delimiters (|).

With `toString`, an **external** representation can be obtained. This representation corresponds to what the third constructor is expecting.

The `clone` method returns a **new copy** of the list, with its own `Vector` of elements. The instances of `Month` are not copied, since they cannot be changed.



Jan 2001

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1 ☑	2 ☑	3 ☑	4 ☑	5 ☑	6 ☐
7 ☐	8 ☑	9 ☑	10 ☑	11 ☑	12 ☑	13 ☐
14 ☐	15 ☑	16 ☑	17 ☑	18 ☑	19 ☑	20 ☐
21 ☐	22 ☑	23 ☑	24 ☑	25 ☑	26 ☑	27 ☐
28 ☐	29 ☑	30 ☑	31 ☑			

Mar 2001

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1 ☑	2 ☑	3 ☐
4 ☐	5 ☑	6 ☑	7 ☑	8 ☑	9 ☑	10 ☐
11 ☐	12 ☑	13 ☑	14 ☑	15 ☑	16 ☑	17 ☐
18 ☐	19 ☑	20 ☑	21 ☑	22 ☑	23 ☑	24 ☐
25 ☐	26 ☑	27 ☑	28 ☑	29 ☑	30 ☑	31 ☐

Figure 8.15: An HTML form for the selection of days (from a list of months previously selected in a similiar way).

Usual **list manipulation** methods are provided: `addElement` appends a `Month` at the end of the list, `addElementSorted` inserts it in chronological order, `size` returns the number of elements in the list, `elementAt` returns the ith element, `contains` checks if a `Month` is present in the list, `remove` receives a `Month` and deletes the first corresponding instance from the list.

Two lists can be **compared** using method `equals` which returns true if both lists contain the same months.

With `getDays` a new `DayList` containing all the days in all the months is computed.

8.2.8 Interface and control classes

In addition to the entity classes described in the previous sections, the package "calendar" contains a few classes that are able to provide visual representations of calendars, either as HTML tables or with widgets of the standard Java library (AWT). Instances of theses classes correspond to different time granularities (years, months, days) and can be used to store, display and modify the status (selected or unselected) of the corresponding elements.

An example of such visual representation is visible in Fig. 8.15, p. 160. A `DayList` is used to store selected dates (those with a checkbox). Interface

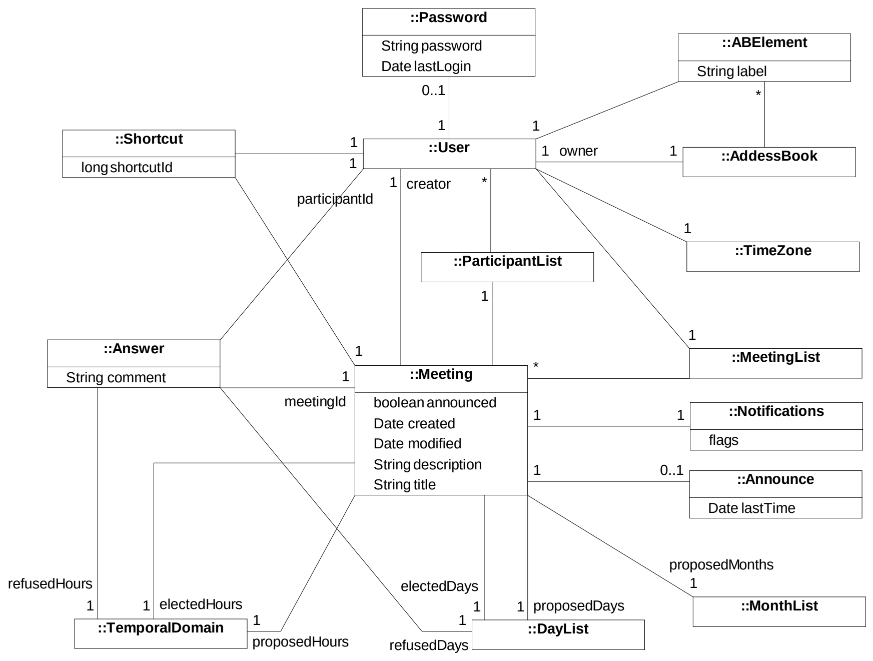


Figure 8.16: UML class diagram representing the main entities at a conceptual level. While it is quite close to the classes that are actually implemented it doesn’t represent them exactly. Some of the actual Java classes are missing and the dependences on this model don’t represent actual dependencies or associations between instances. The entities in this diagram are closer to the tables of the database and their relationships, which are stored using long integers.

classes perform the necessary computations to represent the calendar in HTML format, with the right set of boxes checked. They also examinate the answer sent by the client, detect which boxes are still checked and update the `DayList` correspondingly.

8.3 Package “meety.service”

In addition to the calendaring classes above, the “internal” interface of the service is made up of classes in package “meety.service”. The diagram in Fig. 8.16, p. 161 depicts these classes at a conceptual level.

By following the sequence of the use-cases, we can briefly describe the responsibilities of these classes :

Obtain password: The class `User` is responsible to represent a user, with a unique user ID, and an email address. The class `Password` stores a registered user's password and remembers the last time the user logged in, such that unused accounts can be cleared after a few months.

Manage meeting list: For each `User`, a `MeetingList` is used to store references to the meetings in which the user is involved, either as creator, or as participant. For each `Meeting`, a title, a description, a reference to the creator are stored, as well as the creation and last modification dates.

Create meeting: For each `Meeting`, the system maintains a `Participant-List` that stores references to invited users. A `MonthList` is used to stored proposed months, a `DayList` stores dates proposed by the organizer, a `TemporalDomain` can be used to store possible stretches of time with a finer granularity. An instance of `Notifications` is a means for the system to know which types of notifications are enabled for a given meeting. Current types of notifications are (1) the organizer receives an email every time a participant answers, (2) the organizer receives an email when all participants have answered, (3) the organizer receives an email when all possible dates have been removed by participants, (4) the system resends invitations to participants that have not answered every 24 hours. Class `Announce` is used to determine whether invitations have already been sent. It remembers at what time the announcement was made such that the delay of 24 hours between two notifications can be respected.

Answer to invitation: Email messages sent to invite participants contain an URL that can be used to provide an answer, even for participants that are not registered users. The URL is different for each participant and corresponds to a unique `Shortcut`. Since each shortcut is associated with one `User` and one `Meeting`, the system is able to figure out which information must be sent to the client that uses this URL. It is usually an HTTP form that the participant can use to provide his `Answer`. The `Answer` stores dates and times refused by the user and an optional comment.

Update constraints: *No additional entities.* Display all `Meeting`s in the current user's `MeetingList` whose proposed days or times intersect a given `TemporalInterval`.

Choose time: First, all the `Answers` associated to the current `Meeting` are summarized, then the organizer schedules the `Meeting`: he specifies the value of "electedDays" (a `Daylist`) and optionally of "electedHours" (a `TemporalDomain`).

Change password: *No additional entities.* A new `Password` is associtated with the current `User`.

Merge accounts: *No additional entities.* All the entities formerly associated with a given `User` are associated with another one. This requires cautious handling of some special cases (duplicates in `Shortcuts`, `MeetingList`, etc.).

Manage address book: For each `User`, the system stores an `AddressBook` that can be used to retrieve his acquaintances (`ABElement`) and select meeting participants among them.

In addition, the system associates a `TimeZone` with each `User` in order to display correct temporal informations. For instance the creation date of a meeting, that must not be formatted according to the time zone of the server, but according to the time zone of the user.

Other important classes in package "meety.service" are `Facade` and `Authorization` which have already been presented in § 3.4.4, p. 63. A single instance of `Facade` is instantiated and publised in Voyager's `Namespace`. The `Namespace` plays an equivalent role to the global dictionary in `MØ` (§ 3.5.2, p. 69). Examples are provided in Chap. 9.

8.4 Package "meety.core"

This package contains all the classes that must not be accessed directly by clients, mostly database managers. Our implementation uses MySQL[1], an open source relational database management system. Communication between Java and the database uses JDBC (Campione et al., 1999) and "MM MySQL" open source JDBC drivers[2]. Principles for the implementation of managers have already been described in § 3.4.2, p. 61.

One peculiarity of our implementation is that address books are not stored in the same database as other entities but are managed on their own by an "external" service (Greppin, 2000). Furthermore, this external service is not

[1]http://www.mysql.com/
[2]http://mmmysql.sourceforge.net/

implemented with Java technology but with PHP Web technology[3]. There was absolutely no technical reason to split the service, in fact, it made the implementation more complicated, but it was interesting to work with two different technologies in order to compare them. It was also informative to face the concrete problems of physical distribution, to see how sessions can be managed when more than one service is involved, etc. Additionally, it makes sense to isolate address management from the rest of the service because this part may be reused for other Web services that also require an address book for each user. The address book service can be accessed by HTTP. A few conventions enable (1) a session that starts when Meety authenticates a registered user with his email address and password to be safely "inherited", (2) requests coming from Meety to be authenticated and trusted, knowing that Meety itself doesn't reveal confidential informations to unauthorized users. Only small modifications of the address book service are necessary, in order to extend these conventions and trust additional third-party services.

Chapter summary

The current implementation of Meety follows the principles of Chap. 3. Classes in the interface provide access to meeting informations and events at a procedural level. Additional classes, isolated in their own package produce HTML pages and handle user input. Other kind of interfaces, specific to some devices or services can supplement the existing ones, without collaboration between clients and the service provider, thanks to the ability to move code.

The classes in package "calendar" that have been presented in detail in this chapter represent a potentially useful addition to standard Java calendaring. These classes are independent from Meety and could be reused by other applications that must deal with temporal data.

The methods of all classes in the service software interface are public. We expect to be able to preserve their signatures, even if the internals of the service must be changed, and the implementation of the methods must be rewritten accordingly. In an open context, where many clients from unknown origins may decide to interact with the service, it is extremely important to ensure this kind of stability. The best way to avoid introducing changes that break the compatibility with existing clients is to consider interface classes and method signatures as contracts and to exploit encapsulation to preserve these contracts. This necessity to offer a stable interface is an additional motivation to keep service interfaces very simple, and to let the clients extend them, by executing their own strategies directly on the server. This is much better that

[3]http://www.php.net/

cluttering the service with features that are specific to one client, that become obsolete when the client changes or disappears, and that make the service much more difficult to maintain and to understand.

From the point of view of constraint satisfaction, meetings can be considered as variables and their domains are the dates or temporal domains proposed by the organizer. The answers of participants represent constraints, that the organizer is able to visualize using our system. We found that no great sophistication is necessary in order to present constraints in an easy to understand format, and to let the organizer use his own judgement to select the best date for the meeting. Since the system doesn't actually choose the values assigned to variables, and doesn't try to reschedule when a meeting is over-constrained, it avoids the combinatorial complexity that usually strikes in this kind of programs.

Dependencies between meetings are also out of the scope of the system. Such dependencies (people attending two meetings, need to finish one meeting before the next one starts as in job scheduling problems, etc.) are usually difficult to formalize outside of a specific organizational context, hence organizers and participants with their understanding of their specific context are much better positioned to avoid conflicts and ensure consistency.

Unlike previous systems based on mobile code, which tried to exploit mobility and insisted on the mobile parts of their systems, our implementation has no mobile parts! So far, we have concentrated our efforts on describing the architecture of services, and on the mechanisms that are necessary to enable mobile entities, but not on the entities themselves. It is only in the next chapter that this dissertation focuses on mobile extensions.

Chapter 9

Extensions

Chapter highlights

- A middleware product like Voyager offers convenient high-level operations: we show how easy it is to start a platform within a Java program, to publish and retrieve Meety's `Facade`, and to start an extension on a remote platform.

- Extensions are mobile objects, they encapsulate protocols chosen unilaterally by the extension programmers: however, we also show that there are a few practical considerations which do require prior conventions between service providers and extension programmers.

- Interaction with a remote extension, after a disconnection: this doesn't require conventions between service provider and extension programmer, it is sufficient to instruct the extension how to publish itself in the platform's `Namespace`.

- Preserving the state of extensions when the service is stopped and restarted (aspect of infrastructure dynamicity): this does require a convention of the form "publish and subscribe", in which extensions inform the service that they are present.

- Storing exception that occurs on the server side, while the client is disconnected: doesn't require conventions, the extension stores exceptions chronologically and sends them to the user the next time he reconnects, according to a protocol chosen unilaterally by the extension programmer.

- Application of all the preceding mechanisms in a real extension that sends reminders to participants the day before a meeting: shows how Meety's `Facade` can be used to satisfy a new specific need.

9.1 Extensions as Voyager mobile agents

Extensions are additional functionalities that haven't been implemented by the service provider. Instead, they are developed to satisfy specific needs of a particular user or of an external application. They are especially useful to combine new systems with our service, and to maintain the coherence of data inside it. To obtain such an extensible system, we have exploited the fact that mobile code enables protocol encapsulation and disconnected operation.

Using Voyager's support for "mobile autonomous agents" (ObjectSpace, Inc., 1999), it is not difficult to implement extensions and to install them on the running service. The Voyager platform on which Meety is running can be configured to run agents coming from any Internet host that runs its own Voyager platform. Furthermore, the service publishes a `Facade` in the platform's `Namespace`[1], such that extensions can access important data and events directly as Java objects (Fig. 8.1, p. 139).

In the following examples, we will assume that the Meety **service** is running as a Java application on host cuisun25.unige.ch, and that it has started a Voyager platform that listens on port 8000. With just two lines of Java source code, a **client** Java application can start its own Voyager platform, and obtain a remote reference to the `Facade`:

```
// Code executed by a client of Meety to access the service.
// Start a local platform, that will listen on port 8001:
Voyager.startup(this, "8001");
// Obtain a remote reference to Meety facade, at the default location:
IMeetyFacade facade = (IMeetyFacade) Namespace.lookup(
  "//cuisun25.unige.ch:8000/MeetyFacade");
```

This example shows how middleware products like Voyager are able to hide the complexity of distribution and communication. Once a remote reference has been obtained, it can be used either to call the methods of the `Facade`, or to specify the destination of an agent like in the following code fragment:

```
// Code executed by a client of Meety to send an extension.
// Obtain the agent facet of an existing "extension" object:
IAgent agt = Agent.of(extension);
// Specify where the classes of the extension can be loaded by the service:
```

[1]Package "com.objectspace.voyager".

```
URL dir = new URL("http://cui.unige.ch/~queloz/voyagerlib/");
agt.setResourceLoader(new URLResourceLoader(dir));
// Move the extension and execute method "start" in a remote thread
// ("facade" is the remote reference from the previous code fragment):
agt.moveTo(facade, "start");
```

Within the context of Voyager, the designation “mobile autonomous agent” is in fact just a fancy name for a mobile object with its own thread of control. There are a few restrictions on the objects that can thus be moved. In this example, `extension` must be serializable, the methods of its class must be synchronized, and all its attributes must be serializable. Transient attributes are not copied. Attributes that must be kept but are not serializable must be stored as instances of `Proxy`[2]. Mobile objects must be created with Voyager's object `Factory`[3] and not `new`, Java's default instantiation command. When the object is gone, a proxy stays and transmits next messages. A mobile object is garbage-collected when there are no more references. The proxy is not counted as a reference. Moreover, the class may implement interface `IMobile`[4] in order to supply methods that Voyager invokes before and after the object is moved.

When the agent is autonomous it is not garbage-collected even if there are no more references (local and remote). This is the default behavior and an agent that wants to be destroyed must call `IAgent.setAutonomous(false)`. The agent itself may call `moveTo`, but only the attributes are kept: the stack, local variables and program counter are lost and the agent resumes with the designed callback. Thus, within an agent, there must be no code after such an instruction (only exception handling code).

9.2 Interacting with an extension

Some mobile agent systems, like Hive (Minar et al., 1999), Aglets (Lange and Oshima, 1998) or JumpingBeans (Ad Astra Engineering, Inc., 1998) provide a GUI to track an agent that is moving around the network. In Voyager, there is no such facility, interaction with an agent can occur exclusively by method invocation. However, interacting with the extensions is quite easy, since Voyager is able to hide the fact that an object is not located on the same Java virtual machine by wrapping proxies around it. When the objects (agents) move around a network of inter-connected Voyager platforms, the proxies are able to track their location and to forward method invocations to

[2]Package “com.objectspace.voyager”.
[3]Package “com.objectspace.voyager”.
[4]Package “com.objectspace.voyager.mobility”.

the right place. This concept is called **location transparency** and allows the interaction with an extension even if it has been moved several times.

But the mechanism can work only while the Voyager platforms remain connected to each other. A platform that is stopped and restarted loses all its proxies and is not able to send messages to remote agents any more. In our case, we advocate the use of mobile code for disconnected operation, and thus we require a mechanism that lets users install an extension, disconnect their computer from the network and reconnect later to check the status of the extension or remove it from the system.

One possible way to handle this problem is to let the extension register itself in the `Namespace` of the service's Voyager platform. Thus it can be retrieved from the client's platform when it is needed, just like the `Facade` was retreived in the first code fragment:

```
// Code executed by an extension to become accessible from anywhere
// (findMe is an attribute of the extension, of type String):
Namespace.bind(findMe, this);
```

With this instruction, the extension is published in the `Namespace` and can be retrieved by the client, even if it was stopped and restarted. The name that is assigned to the extension can be chosen unilaterally by the client, without any collaboration with the service. Because the `Namespace` cannot be browsed, it is not possible for someone who doesn't know this name to access the extension. It may be possible to guess the name of an extension, but a long `String` of arbitrary characters can be used to make the probability of a good guess very low.

When the client has obtained a proxy of the extension, using the method `lookup` of `Namespace`, interaction with the extension can simply occur by remote method invocation. One additional precaution is necessary to ensure that the client receives a **remote reference** to the extension and not a **copy**. Indeed, all extensions must be serializable in order to make their first "jump" to the service, and Voyager passes serializable objects by value, not by reference. But if the extension implements the interface `IRemote`[5] Voyager knows that the extension must not be passed by value, and it is still able to move it when the `Agent` facet is used.

This mechanism is illustrated by the three simple Java classes in Fig. 9.1, p. 171.

- `IGoOnDemand` is an interface that specifies which methods the extension is able to respond to. It is important, because the instances returned by Voyager (local or remote) can be converted to this type, and hence the

[5]Package "com.objectspace.voyager".

three methods can be called. It requires (1) a method `kill` to remove the agent from the remote `Namespace`, (2) a method `start` to activate the agent on the remote platform and to register it in the remote `Namespace`, (3) a method `status` to check the status of a remote agent.

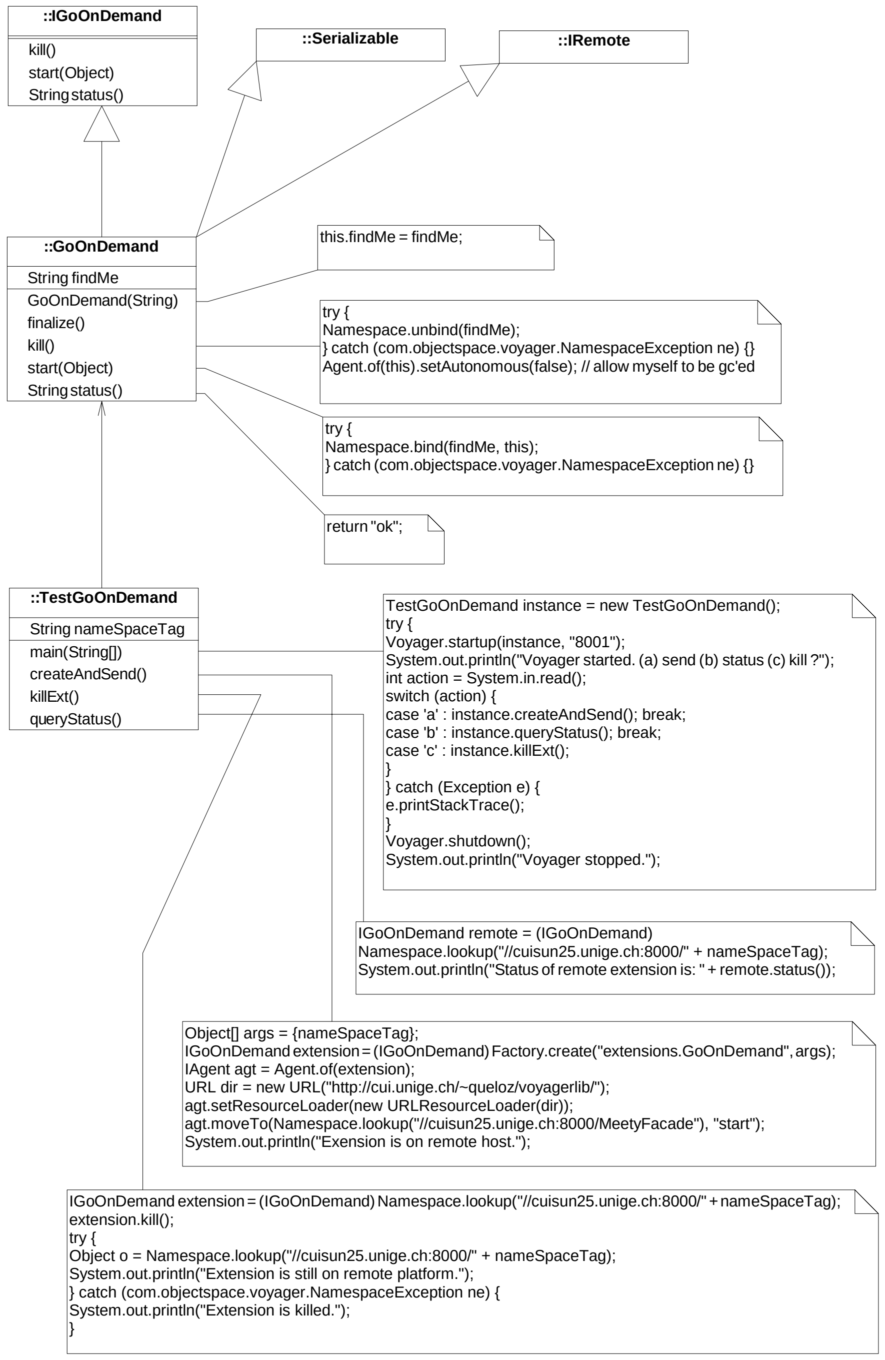
- The second class is `GoOnDemand`, the extension itself. It implements `Serializable`, thus Voyager knows that the instances can be moved. It also implements `IRemote`, to indicate that it must be returned by reference and not by value. Last it implements `IGoOnDemand`, thus the proxies returned by Voyager are compatible with this interface by default (same name, prefixed with letter "I"). Since the attribute `findMe` is not transient, it keeps its value when the instance is moved.

- The third class is the client, that is able to send the extension to our service (`createAndSend`), to interact with it while it is located close to the service (`queryStatus`) and to ask it to unbind from the `Namespace` in order to allow its destruction. This is the best way to destroy an agent, since it may have established connections with many other remote objects and also because Voyager doesn't provide an agent destruction mechanism, unless the client keeps the facet that was used to move the agent (`agt` in `createAndSend` method, which is lost when the client is stopped).

Although it is very simple, this example also exploits and illustrates the essential property of mobile code: that the client is able to choose a protocol unilaterally, according to its needs, despite the extension is actually running on the server. Here the status and termination operation are very simple, but it doesn't require any changes to the service to program much more complex extensions, and to have for instance an extension that moves back to the client's platform when its task is completed or on request.

9.3 Service shutdowns

It is necessary to handle the case where the service (its Java Virtual Machine with the Voyager platform) is stopped, because there is additional **state** brought by extensions that must not be lost. In (Muhugusa, 1997) this is defined as a **down event** and a set of conventions to handle such events (and the complementary **up events**) are described. These conventions allow that a distributed service preserves or updates its state in case of down events. In our case, the distributed service corresponds to one client with the extensions it has installed on the platform to be stopped.



```
this.findMe = findMe;
```

```
try {
Namespace.unbind(findMe);
} catch (com.objectspace.voyager.NamespaceException ne) {}
Agent.of(this).setAutonomous(false); // allow myself to be gc'ed
```

```
try {
Namespace.bind(findMe, this);
} catch (com.objectspace.voyager.NamespaceException ne) {}
```

```
return "ok";
```

```
TestGoOnDemand instance = new TestGoOnDemand();
try {
Voyager.startup(instance, "8001");
System.out.println("Voyager started. (a) send (b) status (c) kill ?");
int action = System.in.read();
switch (action) {
case 'a' : instance.createAndSend(); break;
case 'b' : instance.queryStatus(); break;
case 'c' : instance.killExt();
}
} catch (Exception e) {
e.printStackTrace();
}
Voyager.shutdown();
System.out.println("Voyager stopped.");
```

```
IGoOnDemand remote = (IGoOnDemand)
Namespace.lookup("//cuisun25.unige.ch:8000/" + nameSpaceTag);
System.out.println("Status of remote extension is: " + remote.status());
```

```
Object[] args = {nameSpaceTag};
IGoOnDemand extension = (IGoOnDemand) Factory.create("extensions.GoOnDemand", args);
IAgent agt = Agent.of(extension);
URL dir = new URL("http://cui.unige.ch/~queloz/voyagerlib/");
agt.setResourceLoader(new URLResourceLoader(dir));
agt.moveTo(Namespace.lookup("//cuisun25.unige.ch:8000/MeetyFacade"), "start");
System.out.println("Exension is on remote host.");
```

```
IGoOnDemand extension = (IGoOnDemand) Namespace.lookup("//cuisun25.unige.ch:8000/" + nameSpaceTag);
extension.kill();
try {
Object o = Namespace.lookup("//cuisun25.unige.ch:8000/" + nameSpaceTag);
System.out.println("Extension is still on remote platform.");
} catch (com.objectspace.voyager.NamespaceException ne) {
System.out.println("Extension is killed.");
}
```

Figure 9.1: An accessible extension and a client.

The main reasons to stop and restart the server are maintenance and failures. Our hypothesis is that the shutdowns are foreseeable, i.e. that the server runs for a couple of minutes before it actually terminates. This takes into account maintenance cases, and interruptions in power supply (assuming the presence of batteries). Immediate, unexpected shutdowns are much harder to handle, require redundant hardware and very complex protocols and were not considered in our work.

To cope with this part of our system's **infrastructure dynamicity** aspect we must find a way to preserve the state of extensions installed on the platform. This is necessary if we want to alleviate a user's workload with extensions and not impose them the additional burden of checking that the extensions are still installed and configured properly. Thus, we have defined a way to save the state of extensions before shutting down the service, and to restore it when the service is restarted. Even if the client has cut all connections with our service when the event occurs, we guarantee that informations acquired by the extension and stored in non-transient attributes will not be lost.

When the extensions are Voyager autonomous agents, it is fairly straightforward to capture their attributes, to move them to a "backup" platform (e.g. in another city where electrical power is still available) and to restart them. During this operation, only the non-transient attributes of the extension are preserved; transient attributes, but also the stack, the current execution point and the local variables are all lost, according to the **weak mobility** paradigm (Fuggetta et al., 1998). This has three implications for the design of extensions:

1. all the information they want to keep must be stored in non-transient, serializable attributes
2. their methods should be synchronized to avoid a move while a critical section is executing
3. their methods must execute quickly to avoid delaying the move operation[6].

To know which extensions must be sent to the backup platform, the service uses a simple mechanism based on the principle of **publish and subscribe**. It provides a list where the extensions may register themselves, and when it knows the time has come to move the extensions, he browses the list and transfers each "agent" to another Voyager platform. Only the extensions that have voluntarily subscribed to this state preserving mechanism are moved. Others are simply destroyed when the Java process terminates.

The list is an instance of `AgentList` which is wrapped in an instance of `AgentListWeak` (Fig. 9.2, p. 174). The `AgentListWeak` is published in the

[6]Voyager provides a convenient `Timer` class that provides a way of waiting for an arbitrary duration without suspending a thread and staying in a method for too long (ObjectSpace, Inc., 1999).

`NameSpace` with label "MeetyAgentList" (see the constructor of `MeetyRedundancy`). The purpose of the wrapper is security: it removes the possibility to browse the agent list, thus it cannot be used by an hostile program to obtain a reference to the extensions that have registered. Meanwhile, `MeetyRedundancy` is able to browse the "internal" `AgentList` for its intended purpose: knowing which extensions want to be preserved (see method `canMoveAgents`).

The interface `IRestartable` must be implemented by extensions that want to register in the `AgentList`. Thus the service knows that the extensions provide at least a method `restart` (invoked after the agent has moved) and a method `restartFailed` (invoked if the agent cannot be moved). Both these methods are called in method `moveAgent`.

When an instance of `IRestartable` is registered with method `addElement`, a new random key (of type `Long`) is returned. This key can be used later on to unregister the instance with method `remove`. The use of a secret key guarantees that an extension cannot be unregistered by malevolent third parties, as long as the key is not revealed.

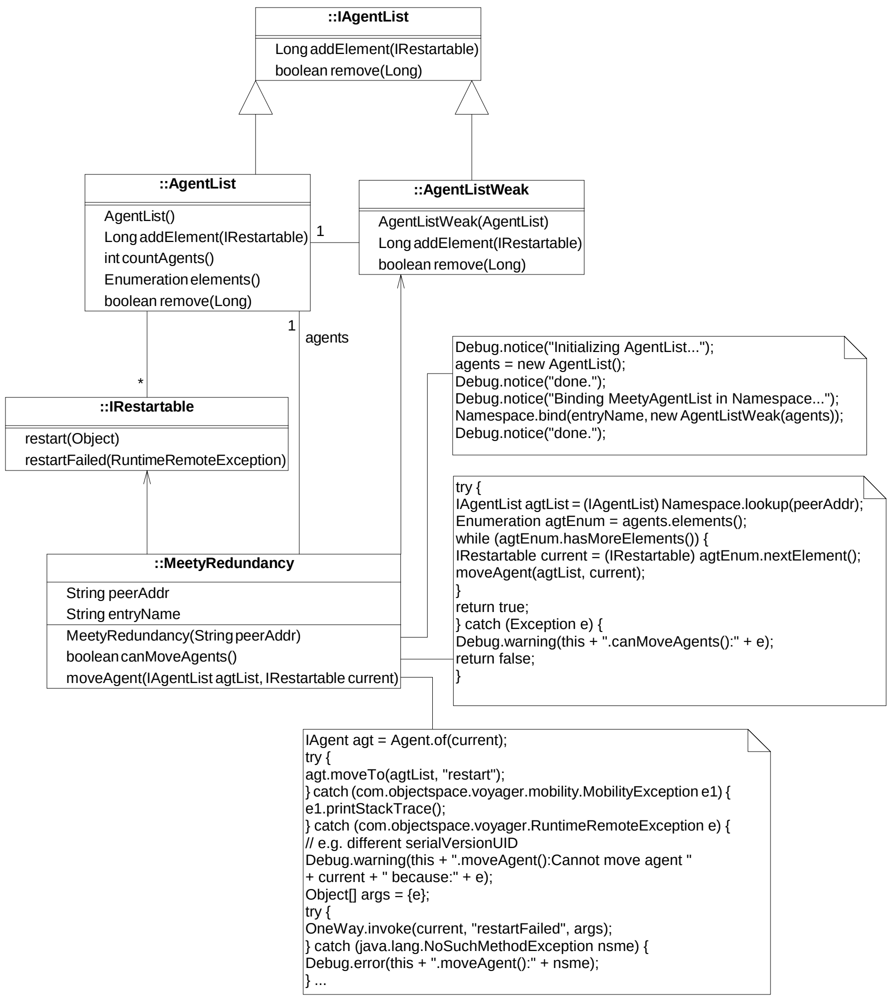
Since its `restart` method is called after the move, the extension knows that something has occurred and it is able to take the appropriate actions. If it was part of a larger system, it might be useful to inform the remaining parts that the extension has moved. The extension is also able to check whether Meety is available at the new location, and may want to register in the local `AgentList` in order to be moved when the backup platform is stopped in its turn.

Removing an object from a platform in such a "brutal" manner may probably be a bad idea, if the platform didn't cease to exist shortly thereafter. It would probably be better to let the extensions tidy up before they leave. But in this case, since the platform is stopped, we know that all the objects that aren't moved will be lost and informing the extensions after they have been moved seems to be sufficient.

If it is not possible to move the extension before stopping the platform, `restartFailed` is invoked to inform the agent that it is going to disappear. In `MeetyRedundancy.moveAgent` a "one-way" method invocation is used to avoid being stuck in the `restartFailed` method of one of the extensions[7]. This guarantees that the platform can terminate. The current strategy is to wait for a few seconds before terminating, thus the extensions have a chance to move to another platform on their own initiative, to send their state through a network connection, or to send an email to their owner.

Our scheme doesn't automatically handle the "up events" that occur when a platform that was temporarily shut down is running again. This means that the extensions are not informed when they may come back to the "pri-

[7] `OneWay` is another convenient utility class provided by Voyager (ObjectSpace, Inc., 1999).



```
Debug.notice("Initializing AgentList...");
agents = new AgentList();
Debug.notice("done.");
Debug.notice("Binding MeetyAgentList in Namespace...");
Namespace.bind(entryName, new AgentListWeak(agents));
Debug.notice("done.");
```

```
try {
IAgentList agtList = (IAgentList) Namespace.lookup(peerAddr);
Enumeration agtEnum = agents.elements();
while (agtEnum.hasMoreElements()) {
IRestartable current = (IRestartable) agtEnum.nextElement();
moveAgent(agtList, current);
}
return true;
} catch (Exception e) {
Debug.warning(this + ".canMoveAgents():" + e);
return false;
}
```

```
IAgent agt = Agent.of(current);
try {
agt.moveTo(agtList, "restart");
} catch (com.objectspace.voyager.mobility.MobilityException e1) {
e1.printStackTrace();
} catch (com.objectspace.voyager.RuntimeRemoteException e) {
// e.g. different serialVersionUID
Debug.warning(this + ".moveAgent():Cannot move agent "
+ current + " because:" + e);
Object[] args = {e};
try {
OneWay.invoke(current, "restartFailed", args);
} catch (java.lang.NoSuchMethodException nsme) {
Debug.error(this + ".moveAgent():" + nsme);
} ...
```

Figure 9.2: Classes used by an extension to subscribe to the “state preserving” mechanism.

mary" platform that runs the Meety service. However, since the extensions are active when they are on the backup platform, they are able to check periodically if the primary platform is running. But we don't like this solution that wastes resources and necessitates an additional effort from the extension programmer. Instead, we propose to send the extensions back to the primary platform once it is running again simply by stopping the backup platform. This operation is symmetric to the first one and can be performed by the same `MeetyRedundancy` class, configured with a valid address (`peerAddr` attribute). Additionally, shutting down the backup platftorm ensures that no objects are left behind. The resulting sequence of events is the following:

1. The primary platform is running, extensions that want to be preserved have registered in the `AgentList`.
2. The backup platform is started, it provides its own `AgentList`.
3. The primary platform is stopped, before it really terminates, extensions are moved to the backup platform, where they register in the `AgentList`.
4. The primary platform is started, it provides `MeetyFacade` and an `AgentList`.
5. The backup platform is stopped, before it really terminates, extensions are moved to the primary platform, where they can register in the `AgentList` and continue to work with `MeetyFacade`.

We think this illustrates that object mobility is a very convenient way to cope with the infrastructure dynamicity aspect: it allows the preservation of important state and the notification of concerned components during the necessary shutdowns of the platforms, with only very little effort from the service provider and extensions programmers.

There are however two additional constraints in this approach. The first one is that the Web server which provides the classes required by the extensions must be responding at steps 3 and 5, otherwise the destination platform is not able to restart the agent. The second one is also related to class loading. The extension cannot be moved at step 3 or 5 if the classes that were instantiated when the agent arrived on the platform and the classes returned by the Web server when it tries to leave are not exactly the same. An exception is thrown as soon as the classes have different version numbers[8].

This versioning problem is related to another typical problem in mobile code environments. When Voyager loads the bytecode of a new class for the first time, it keeps a copy in a cache. Next time the class must be instantiated, it is not reloaded but the cached copy is used. Obviously, this improves performance, but causes trouble when an extension programmer wants to modify his

[8]The attribute `serialVersionUID` is supposed to help the programmer cope with this issue, but we didn't manage to obtain compatible versions of the classes in our test environment, even when the attribute was set in the source code and remained unchanged.

code and restart the extension. In this case, a mechanism to clear the cache is needed, but we didn't manage to find it. The only solution was to give different names to new versions of the extension and other modified classes. In the case of `MØ`, the platform doesn't perform any caching. The code must be sent with each messenger, and this versioning problem doesn't occur. On the other hand, clients that want to improve performances and avoid resending the code every time must implement their own caching mechanism. But this is a better solution since the client actually has the possibility to control which version is used.

This example reinforces our motivation to give as much control as possible to the clients, in this case the extension programmers, because an intensive use will always lead to unexpected cases.

Other approaches

Another way to preserve the state of extensions during shutdowns is to serialize them and to keep them on persistent storage. However, this solution is more complicated for the service provider, who needs to administer the database, and explicitly handle some parameters that are handled implicitly by the agent facet (like the resource loader which needs to be set only once). For the extension programmer, this is also less comfortable, because the agent is unreachable and unable to work during an unpredictable period of time. With our approach, the agent is still "alive" and potentially useful, even if it is not on the primary platform during this period. Moreover, the extension is moved by the service, but there is no loss of control from the point of view of the extension programmer, since the code executed on the remote platform also belongs to the client.

The conventions presented in the work of Muhugusa (Muhugusa, 1997) are also based on the principle of "publish and subscribe": instead of actively polling the platforms until an interesting event occurs—even if it is simple and doesn't require additional conventions or protocols, it requires too much resources—services that need to be notified inform the platform on which they reside by implementing a special "down" procedure. Before shutting down, the platform executes all visible "down" procedures, in which the services perform the necessary cleanup operations. Thus the platform doesn't move the extensions itself, and the "down" procedures are fully responsible to take appropriate actions. So the work is left to extension programmers or third party services. This approach also supposes that other platforms are available, where the extensions can move in order to survive, or that they are part of a vast "distributed service" which can be contacted when the down event occurs. Clearly, these two hypotheses are incompatible with our goal to support disconnected operation. Additionally, executing the "down" procedures provided by the services cannot be done without precautions: there is always a risk that a thread remains stuck in an improperly programmed procedure. With our approach, there is automatically one thread for each agent and if an extension is ill-behaved it cannot prevent the transfer of other extensions or the termination of the platform.

One last remark is that we have applied our scheme to extensions, because they don't have access to persistent storage and need another way to survive platform shutdowns. But it is by no means restricted to extensions and could be used to preserve other parts of the service as well, provided that they follow the same programming conventions.

9.4 Exceptions and debugging

Exceptions may occur when the extension is located on the server. Extension programmers need a description that is as accurate as possible to understand what happened, and to determine if there is a fault in their code and how it can be corrected. The simplest solution, which may be sufficient in a non-distributed setting is to send error messages to a console or to a log file that the programmer can read. Unfortunately, these channels cannot be used to watch extensions located on the server.

Voyager messaging abilities, remote method invocation or even a basic communication channel (e.g. socket) could be used by the extension to send error messages to the programmer, but not in a "disconnected" setting. Note that the standard Voyager platform doesn't provide means to send email, but Meety offers this service to extensions that are in possession of a valid password. However, it is not possible to rely on this because Meety may be unavailable from time to time, and also because network failures may isolate a platform from the rest of the Internet completely.

The best solution seems to keep error messages within the agent, and to send them when the link with the user is re-established. The two classes in figure Fig. 9.3, p. 178 make this very easy. `ExceptionTrace` is able to store the description of an exception and the time when it occurred. The responsibility of `ExceptionStore` is to keep an ordered list of `ExceptionTrace`s and to remember when it started building the list.

An agent that needs to keep detailed error messages can do it with a single non-transient attribute of type `ExceptionStore`. Since both classes are serializable, their instances can be transported when the agent moves. When an exception is caught, it can be recorded with a call to `addElement`. With `readAndClear`, the extension is able to send the whole list of errors to the user or programmer, using the interaction mechanisms described in § 9.2, p. 168. We have also found that this approach is very convenient to keep

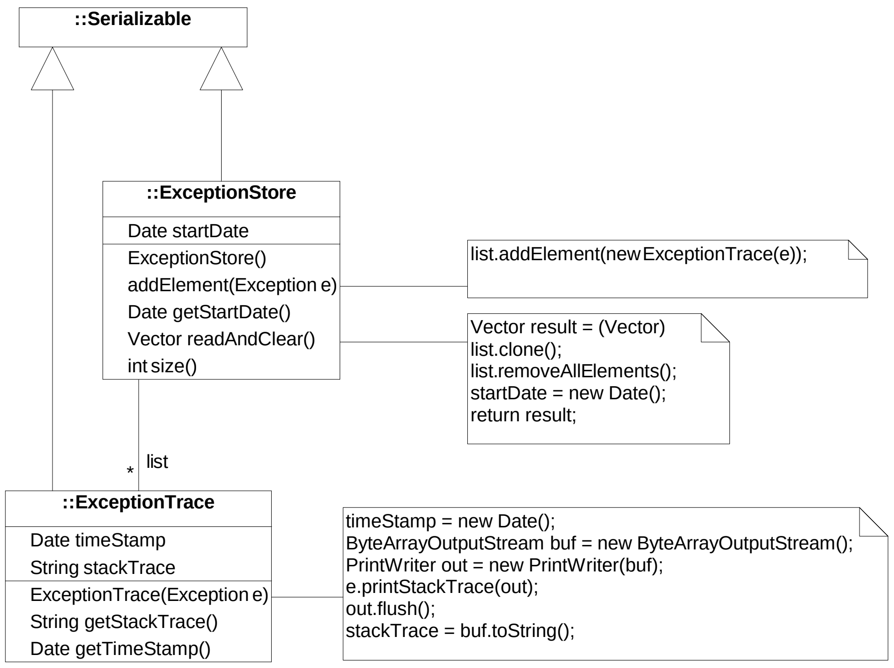


Figure 9.3: Classes used by an extension to keep a description of caught exceptions.

non-exceptional activity logs. To do this, the agent calls `addElement` with any kind of exception instantiated just when a noteworthy event occurs, and both kinds of messages remain temporally interleaved in the `ExceptionStore`.

The worst shortcoming of the current implementation is that the amount of memory to store exception traces is not limited, thus an exception that occurs very often could lead to huge data structures. Such a big list could not only fill the platform's memory, but may also take too long to transfer, either when the agent moves, or when the user wants to see error messages. Simple solutions could be to limit the number of entries in the list, or the total size in bytes. However neither solution is perfect, because they both result in throwing away information that is potentially useful. In any case, this shortcoming is not a security threat in itself, since an agent that wants to fill the memory may do so by allocating any kind of object anyway. A much better approach seems to be resource control at platform level (§ 3.5.5, p. 71).

The final word is that this mechanism is worth presenting because it was sufficient to debug extensions in our case, whereas it is almost impossible to understand why an extension fails without it. And more importantly because it is another illustration of the flexibility offered by mobile code: the extension programmer himself can choose how he wants to handle exceptions, since he provides the code executed when the extensions are running on a remote platform.

9.5 Working periodically

The way we handle exceptions is well illustrated by one simple extension called `PeriodicTask`. This extension is not very useful in itself (it wakes up periodically but doesn't perform any work) but it can be extended by a subclass, as we will show in section 9.6. The UML diagram of this extension is given in Fig. 9.4, p. 180.

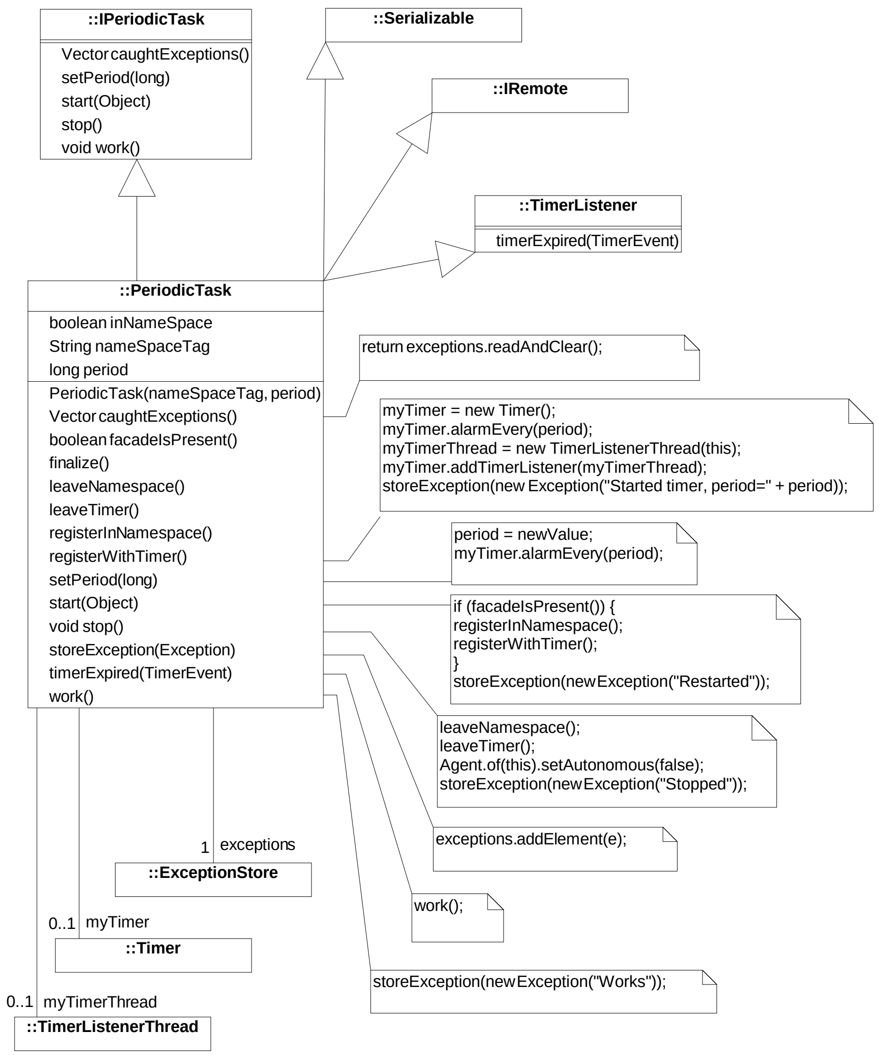
The constructor of this extension takes two parameters which are stored in non-transient attributes. The first one is the name that will be used to locate the extension in the `Namespace` (just like the `findMe` attribute of `GoOnDemand` in Fig. 9.1, p. 171). The second parameter is the period in milliseconds.

To interact with this extension, the methods in interface `IPeriodicTask` must be used. The method `start` must be invoked when the extension is moved to a remote platform. This method checks that Meety facade is present. If it is the case, it makes itself visible in the `Namespace`, then it calls `registerWith-Timer`. Finally, it uses its `ExceptionStore` to keep a trace of the event. The extension does nothing if Meety facade is not available on the local platform, and doesn't register with the state preserving service of section 9.3.

The method `registerWithTimer` illustrates how a `Timer`[9] can be created, and asked to "tick" every `period` milliseconds. The event notification follows the standard JavaBeans event notification model (Sun Microsystems, 1997). Every time the period is elapsed, the `Timer` calls the method `timerExpired` of all registered `TimerListeners`. A `TimerListener` can be added using the method `addTimerListener`. Since `PeriodicTask` implements `TimerListener`, it could potentially be notified directly by the `Timer`, but in the current implementation an additional `TimerListenerThread` is used to decouple the two objects. Wihtout this precaution, the whole management of timers can be thrown into confusion if `timerExpired` doesn't return quickly enough.

Other important methods in `IPeriodicTask` are: `stop` which unregisters the extension from the platform, `setPeriod` for changing the rate of activity, `caughtExceptions` which returns the contents of the `ExceptionStore` and

[9]Package "com.objectspace.lib.timer".



```
return exceptions.readAndClear();
```

```
myTimer = new Timer();
myTimer.alarmEvery(period);
myTimerThread = new TimerListenerThread(this);
myTimer.addTimerListener(myTimerThread);
storeException(new Exception("Started timer, period=" + period));
```

```
period = newValue;
myTimer.alarmEvery(period);
```

```
if (facadeIsPresent()) {
registerInNamespace();
registerWithTimer();
}
storeException(new Exception("Restarted"));
```

```
leaveNamespace();
leaveTimer();
Agent.of(this).setAutonomous(false);
storeException(new Exception("Stopped"));
```

```
exceptions.addElement(e);
```

```
work();
```

```
storeException(new Exception("Works"));
```

Figure 9.4: An extension that works periodically and stores exceptions. The main methods, used to control the extension, are those of interface `IPeriodicTask`.

clears it.

The `work` method, which is normally invoked periodically by `timerExpired` can also be invoked "externally" since it is visible in `IPeriodicTask`. This possibility was mainly used for testing, where it is not very convenient to observe a periodic activity.

9.6 A real extension: sending reminders

Now that basic mechanisms have been explained, we can describe a real extension. The responsibility of this extension is to sent reminders the day before a meeting will take place. The reminders are sent by email to the participants. The extension works on behalf of one user and can be controlled by a very simple client program, following the mechanisms of the previous sections.

The UML diagram of this extension is given in Fig. 9.5, p. 182.

In the following presentation, only the methods that are specific to the task are presented. Operations that allow the extension to be contacted by the client after disconnections, to survive service shutdowns, or to store and send exceptions to the client are implemented like in the previous sections and are not repeated.

The constructor `DontForget`, whose source code is presented in the box below, initializes the superclass `PeriodicTask` to run every hour, then it stores the address and the password of the extension's owner in non-transient fields. These informations are required by Meety facade to access meeting informations.

The constructor also initializes the attribute `quietMeetings` which contains a list of meetings for which the user doesn't want to send reminders.

The last step is for the attribute `lastAction` which obtains a value 24 hours before the requested `firstAction`. Afterwards the exception is activated every hour, but it checks meetings and sends reminders only when `firstAction` is already 24 hours in the past. This is the solution we have chosen to have an extension that works everyday at approximately the same time, and that doesn't send reminders more than once.

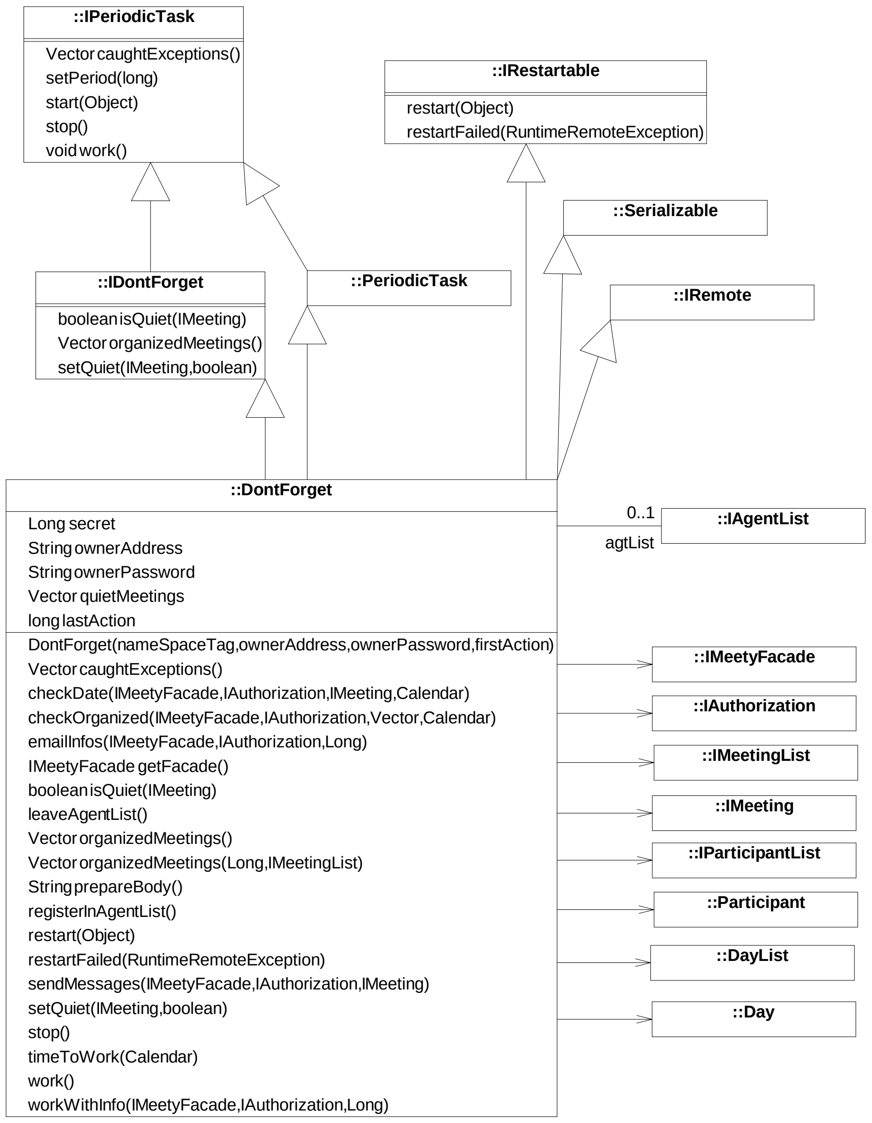

::IPeriodicTask
Vector caughtExceptions()
setPeriod(long)
start(Object)
stop()
void work()
::IRestartable
restart(Object)
restartFailed(RuntimeRemoteException)
::Serializable
::IDontForget
boolean isQuiet(IMeeting)
Vector organizedMeetings()
setQuiet(IMeeting,boolean)
::PeriodicTask
::IRemote
::DontForget
Long secret
String ownerAddress
String ownerPassword
Vector quietMeetings
long lastAction
DontForget(nameSpaceTag,ownerAddress,ownerPassword,firstAction)
Vector caughtExceptions()
checkDate(IMeetyFacade,IAuthorization,IMeeting,Calendar)
checkOrganized(IMeetyFacade,IAuthorization,Vector,Calendar)
emailInfos(IMeetyFacade,IAuthorization,Long)
IMeetyFacade getFacade()
boolean isQuiet(IMeeting)
leaveAgentList()
Vector organizedMeetings()
Vector organizedMeetings(Long,IMeetingList)
String prepareBody()
registerInAgentList()
restart(Object)
restartFailed(RuntimeRemoteException)
sendMessages(IMeetyFacade,IAuthorization,IMeeting)
setQuiet(IMeeting,boolean)
stop()
timeToWork(Calendar)
work()
workWithInfo(IMeetyFacade,IAuthorization,Long)
0..1
agtList
::IAgentList
::IMeetyFacade
::IAuthorization
::IMeetingList
::IMeeting
::IParticipantList
::Participant
::DayList
::Day

Figure 9.5: An extension that sends reminders the day before a meeting occurs. The methods to control the extension are those of interfaces `IPeriodicTask` and `IDontForget`.

```
/**
 * Parameter firstAction is the next time the method must work
 */
public DontForget(String nameSpaceTag, String ownerAddress,
  String ownerPassword, Long firstAction)
{
  super(nameSpaceTag, new Long(3600000)); // run every hour
  this.ownerAddress = ownerAddress;
  this.ownerPassword = ownerPassword;
  quietMeetings = new Vector();
  lastAction = firstAction.longValue() − 86400000;
}
```

The next two methods allow the management of the list `quietMeetings`. They work with the meeting identifier, a `Long` value that is easier to handle than the meeting instances or their proxies. When a meeting is "quiet", no reminder is sent.

```
/**
 * Returns true iff the meeting is listed in quietMeetings
 */
public synchronized boolean isQuiet(IMeeting m) {
  return quietMeetings.contains(m.getId());
}
```

```
/**
 * Inserts or removes m from the list of "quiet" meetings
 */
public synchronized void setQuiet(IMeeting m, boolean dontSendReminder) {
  if (isQuiet(m)) { // in list
    if (!dontSendReminder) {
      quietMeetings.removeElement(m.getId());
    }
  } else { // not in list
    if (dontSendReminder) {
      quietMeetings.addElement(m.getId());
    }
  }
}
```

The next method conveniently returns a collection of all meetings organized by the owner of the extension. This allows for instance that the client program displays all meeting titles in a list, in order to choose which meetings are quiet. The first step is to look for Meety facade in the `Namespace`, then to use the address and password to obtain an authorization, then to obtain the user identifier that corresponds to the address (this information is not confidential and doesn't require an authorization). Then the method obtains the list of all meetings related to the user (this requires an authorization). Only a subset of these meetings are actually organized by the owner of the extension and can be returned. The last filtering step is carried out by the second version of `organizedMeetings` which receives the user identifier and the full list of meetings.

```
/**
 * Returns null if the information cannot be obtained.
 * Returns a possibly empty Vector otherwise.
 */
public synchronized Vector organizedMeetings() {
  Vector result = null;
  IMeetyFacade facade = getFacade();
  if (facade != null) {
    IAuthorization perm = facade.getAuthorization(ownerAddress,
      ownerPassword);
    if (perm != null) {
      Long ownerId = facade.getId(ownerAddress);
      try {
        IMeetingList meetings = facade.getMeetingList(ownerId, perm);
        result = organizedMeetings(ownerId, meetings);
      }
      catch (ch.unige.cui.queloz.meety1.service.AuthorizationException ae) {
        storeException(ae);
      }
    }
  }
  return result;
}
```

The second version of `organizedMeetings` returns a new `Vector` with only the meetings for which the creator's identifier corresponds to the given `ownerId`.

```
/**
 * Returns a new Vector with only the meetings organized by this user.
 */
private Vector organizedMeetings(Long ownerId, IMeetingList meetings) {
  Vector result = new Vector();
  for (int i = 0; i < meetings.size(); ++i) {
    IMeeting current = meetings.elementAt(i);
    if (current.getCreator().equals(ownerId)) {
      result.addElement(current);
    }
  }
  return result;
}
```

The next method is `restartFailed`. This method is called when the state preserving mechanism is not able to move the extension. When this method runs, the extension knows that it is going to disappear. Since the owner of the extension will very likely be disconnected at this time, the best solution is to send him an email message. Authorized users can use Meety facade in order to send an email, thus the first step is to retreive the facade and to obtain an authorization. Then `emailInfos` can be invoked to actually send the message.

```
/**
 * Sends an email to the owner (arg might be null)
 */
public synchronized void restartFailed(RuntimeRemoteException arg) {
  IMeetyFacade facade = getFacade();
  if (facade != null) {
    IAuthorization perm = facade.getAuthorization(ownerAddress,
      ownerPassword);
    if (perm != null) {
      Long ownerId = facade.getId(ownerAddress);
      emailInfos(facade, perm, ownerId);
    }
  }
}
```

The method `emailInfos` prepares the fields of the message and asks the facade to send it. Even if an exception occurs, it is not stored because the extension is going to be destroyed.

```
/**
 * Sends an email to the owner, describing the state of this extension
 */
private void emailInfos(IMeetyFacade facade, IAuthorization perm,
  Long ownerId)
{
  String title = "Extension has been stopped";
  String body = prepareBody();
  try {
    facade.sendMessage(ownerId, perm, ownerAddress, title, body);
  } catch (Exception e) {
  }
}
```

The next method prepares the body of the message sent by a dying extension to its owner. It concatenates all exception traces, with their timestamps.

```
/**
 * Returns a String with all stored exceptions, plus the current status
 */
private String prepareBody() {
  ByteArrayOutputStream buf = new ByteArrayOutputStream();
  PrintWriter out = new PrintWriter(buf);
  out.println("Extension cannot be preserved and has been stopped!\n\n");
  Vector exceptions = caughtExceptions();
  int limit = exceptions.size();
  for (int i = 0; i < limit; ++i) {
    ExceptionTrace t = (ExceptionTrace)exceptions.elementAt(i);
    out.println("-----------------------------------------------"
      + "----------------");
    out.println(t.getTimeStamp());
    out.println("-----------------------------------------------"
      + "----------------");
    out.println(t.getStackTrace());
  }
  out.flush();
  return buf.toString();
}
```

When an extension has been moved by the state saving mechanism, the method `restart` is automatically invoked. This method ensures that the superclass `PeriodicTask` is restarted as well and that the extension is registered in the platform's `AgentList` for the next shutdown.

```
/**
 * Method called to restart the agent after it has moved (IRestartable):
 * Tries to restart the PeriodicTask (super)
 * Asks the platform to preserve the state of this agent
 */
public synchronized void restart(Object rejoined) {
  super.start(rejoined);
  registerInAgentList();
}
```

The method `work` is invoked once per hour and is responsible to send reminders for meetings scheduled during the next day if necessary. Calling the `work` method of the superclass ensures that the event is traced. The facade, an authorization, and the owner identifier must be obtained before the extension can actually perform its task.

```
/**
 * When the extension wakes up, it first fetches some information
 */
public synchronized void work() {
  super.work();
  IMeetyFacade facade = getFacade();
  if (facade != null) {
    IAuthorization perm = facade.getAuthorization(ownerAddress,
      ownerPassword);
    if (perm != null) {
      Long ownerId = facade.getId(ownerAddress);
      workWithInfo(facade, perm, ownerId);
    }
  }
}
```

In `workWithInfo`, the first step is to initialize a `Calendar` with the timezone of the owner. This is important to determine which meetings are scheduled for the next day. Then the method calls `timeToWork` to determine if 24 hours have passed since the last action. If it is the case, the list of meetings organized by the owner is retreived, and `checkOrganized` is invoked to send reminders if necessary.

```
/**
 * If 24 hours elapsed since the last action, retreive the meetings
 * organized by extension owner and check if reminders mut be sent.
 */
private void workWithInfo(IMeetyFacade facade, IAuthorization perm,
  Long ownerId)
{
  try {
    TimeZone tz = TimeZone.getTimeZone(facade.getTimeZone(ownerId, perm));
    Calendar cal = Calendar.getInstance(tz);
    if (timeToWork(cal)) {
      IMeetingList meetings = facade.getMeetingList(ownerId, perm);
      Vector organized = organizedMeetings(ownerId, meetings);
      checkOrganized(facade, perm, organized, cal);
      storeException(new Exception("Meetings checked"));
    }
  } catch (ch.unige.cui.queloz.meety1.service.AuthorizationException ae) {
    storeException(ae);
  }
}
```

The result of `timeToWork` is true if the given `Calendar` object (current time) contains a value that is at least 24 hours after the value of `lastAction`. In this case, `lastAction` is updated.

```
/**
 * Returns true if cal is 24 hour later than lastAction
 * Updates lastAction
 */
private boolean timeToWork(Calendar cal) {
  if (cal.getTime().getTime() > lastAction + 86400000) { // act every day
    lastAction = cal.getTime().getTime();
    return true;
  } else {
    return false;
  }
}
```

The next method loops on all meetings organized by the owner of the extension and passes only those which are not quiet to the method that will actually check when the meetings are scheduled.

```
/**
 * Checks the dates of all meetings in organized that are not "quiet"
 */
private void checkOrganized(IMeetyFacade facade, IAuthorization perm,
  Vector organized, Calendar cal)
{
  for (int i = 0; i < organized.size(); ++i) {
    IMeeting current = (IMeeting) organized.elementAt(i);
    if (!isQuiet(current)) {
      checkDate(facade, perm, current, cal);
    }
  }
}
```

In the method `checkDate`, a copy of the `Calendar` is incremented by one day, and if it belongs to the list of days when the meeting is scheduled then messages are sent to participants.

```
/**
 * Sends messages to participants if "current" is scheduled for the next day
 */
private void checkDate(IMeetyFacade facade, IAuthorization perm,
  IMeeting current, Calendar cal)
{
  try {
    Calendar tomorrow = (Calendar) cal.clone();
    tomorrow.add(Calendar.DATE, 1);
    DayList scheduledDays = facade.getScheduledDays(current.getId(), perm,
      cal.getTimeZone());
    if (scheduledDays.contains(new Day(tomorrow))) {
      sendMessages(facade, perm, current);
    }
  } catch (ch.unige.cui.queloz.meety1.service.AuthorizationException ae) {
    storeException(ae);
  }
}
```

The last method `sendMessages` composes the message and sends it to all participants of the `current` meeting, using the `sendMessage` method in Meety's facade.

```
/**
 * Sends an email (reminder) to all participants of current
 */
private void sendMessages(IMeetyFacade facade, IAuthorization perm,
  IMeeting current)
{
  String title = "Meety - Reminder for:  " + current.getTitle();
  String body = "This message was sent by Meety service to remind you\n"
    +"that the meeting with title:\n\n" + ">\t" + title + "\n\n"
    +"is scheduled for tomorrow.";
  try {
    IParticipantList participants = facade.getParticipants(current.getId(),
      perm);
    int limit = participants.size();
    for (int i = 0; i < limit; ++i) {
      Participant part = participants.elementAt(i);
      String destAddr = facade.getAddress(part.getUserId());
      facade.sendMessage(current.getCreator(), perm, destAddr, title, body);
      storeException(new Exception("Message sent to " + destAddr));
    }
  } catch (Exception e) {
    storeException(e);
  }
}
```

With this last method, we have presented the whole source code of a possible extension of our service. According to our practical experience, this extension is useful: in a couple of cases, participants themselves told us that they could have forgotten the meeting, but that the message sent the day before helped them remember.

From a theoretical point of view, the interest of such extensions is that they take into account the precise needs of a user, without requiring an intervention from the service provider. Furthermore, they are not limited to adapting the service to the needs of a user, but can also be used for the integration with other services. In this case, they can adapt the service to the needs of a new application, for instance by performing appropriate data conversions or event propagation.

Chapter 10

Results and lessons

Chapter highlights

- Which non-functional aspects are handled in the architecture and in the case study: having described the implementation of Meety, we can now ground our promise of a simplified treatment of many non-functional aspects with concrete facts.
- Remarks on the concrete usage of Meety: besides its theoretical interest, the case study yielded a service that is concretely useful for a few people who use it from time to time in order to schedule meetings.

10.1 Treatment of non-functional aspects

For each non-functional aspect of Chap. 2 we now examine if our study revealed new opportunities to handle it, or an easier implementation. We also consider benefits of mobile code that have been previously described in the literature.

10.1.1 Delegation to the Client

In § 3.2, p. 53 we suggested that an interesting effect of the architecture would be to free the service provider from taking certains aspects into account, because mobile code would allow clients to implement their own strategies.

Aspect	**Handled by**
event notification	Making important events visible at the `Facade` level is essential. With our approach, the service provider can use simple mechanisms at object level for notifications and doesn't need to send notifications across the network.
flow control	This aspect usually requires collaboration between the service and the client, because the service must suspend the transfer if the client is saturated. Mobile code allows that the client installs its own objects to handle the transfer, thus the service provider doesn't need to specify and implement this aspect.
memory management	The service provider should ensure that all clients get a fair share of available memory, and the execution environment can help. The service doesn't need to send related information across the network, since the clients can run their extensions on the service's platform, and collect availability data directly.
platform adjustment	The service relies on the ability of clients with special needs to hide themselves behind a mobile extension.
prioritization	The service provider should ensure that all clients get a fair share of available processing time, and the execution environment can help. As with memory availability, the availability of CPU time doesn't need to be sent to the client.
protocol negotiation	Unless the service itself is a client of further services, the service designer's mind can be freed from this aspect thanks to the ability of mobile code to encapsulate protocols.
real-time constraints	Clients can improve service responsiveness with extensions that require less interactions across the network (e.g. a client for the Windows operating system).
serialization	Simple classes are able to serialize and deserialize their state into `String`s; more complex classes may use readily available mechanisms (e.g. implement `Serializable` in Java). Other needs can be handled by extensions, which use the methods of the service interface to obtain "raw" information and process it into whatever representation the client needs.

10.1.2 Benefit from Locality

Having extensions that run directly on the server provides better control and avoids some inherent problems of remote interaction (latency, disconnections, etc.). The aspects for which this new possibility of mobile code is particularly interesting are listed here.

Aspect	**Handled by**
accuracy maintenance	Being closer to the information source, the client is able to detect changes earlier, moreover he is able to program his own strategies to maintain the accuracy of his own data when a relevant event occurs.
bandwidth management	The client can use a mobile extension to control the amount of data sent back, whereas the service usually controls the amount of data sent with reply/request schemes. Mobile code is well known for its ability to save bandwidth.
buffering	Although this aspect is related to communication, it is usually implemented on the client side only, thus mobile code is not essential. However, the buffer could be managed by an extension on the server side, for clients that do not have sufficient memory.
caching	In our case study, caching is used only to avoid querying the database when entity objects have not changed; caching informations exchanged between the service and its clients has not been studied. This aspect cannot be completely delegated to the client, because some conventions are necessary to inform him when the cached data must be invalidated. However mobile code could be useful to control where the cache is located, or to store events while the client is disconnected (§ 11.4.1, p. 204).
coordination	With our architecture the service provider controls concurrent accesses at the `Facade` level and can ensure that there are no internal coordination problems. Recent studies indicate that the ability to co-locate tasks that must be coordinated has a positive impact on workflows (Tripathi et al., 2000), and solves some distributed coordination problems in a robust and efficient way (Rowstron, 1999).
event notification	The extensions are able to react quickly to events, or store them if the client is not reachable at the time the event occurs.
exception handling	Mobile extensions are able to store exception traces until the client is able to view them (§ 9.4, p. 177), this makes debugging of the extensions easier. Mobility also provides the opportunity of distinguishing between service failures and network errors, something that is usually difficult with physical distribution. Some problems related to partial failures can also be avoided (Rowstron, 1999).

continued on next page

continued from previous page

Aspect	**Handled by**
indexing	Several authors have shown that mobile extensions are able to build indexes faster and using much less bandwidth than immobile "robots" (Vigna, 1998, chapter 6), (Brewington et al., 1999), (Sudmann and Johansen, 2000). In the case of Meety, this is not really relevant.
infrastructure dynamicity	Because they are physically co-located with the service, mobile extensions can continue working even if the client is disconnected from the network, and they can easily be retrieved, even if the client has moved when he reconnects (§ 9.2, p. 168). Because they can easily be restarted on another host, extensions can preserve their state, and even remain active when the service is stopped for maintenance reasons (§ 9.3, p. 170).
memory management	The client is able to handle specific situations in accordance with its particular needs, instead of being subject to an inadapted generic mechanism.
prioritization	Like **memory management**, the consumption of CPU time can be controlled in a client specific-way.

10.1.3 Execution Environment

Ideally, a mobile code execution environment should provide much better support for the implementation of services that standard workstation operating systems. Programming distributed applications on these platforms is generally possible but too arduous.

Aspect	**Handled by**
access control	In Meety, the `Facade` checks `Authorization`s and rejects some operations. The execution environment is also often responsible to limit the operations that foreign code may perform (§ 3.5.4, p. 70).
accounting	Not handled by the architecture but essential to control the amount of resources (bandwidth, memory, cpu) consumed by foreign code that runs on the same host as the service. The execution environment can provide precious low-level support (§ 3.5.5, p. 71).
accuracy maintenance	The execution environment may offer additional mechanisms like leasing to handle this aspect (§ 11.1, p. 200).
activation	The architecture exploits the ability to move graphs of objects to a secondary platform to preserve their state without storing them on persistent media (§ 9.3, p. 170).

continued on next page

continued from previous page

Aspect	**Handled by**
authentication	In the case study, the `Facade` grants an `Authorization` when username and password are matching. The origin of foreign code signed with digital certificates can be authenticated and additional access rights can be granted, without compromising security (§ 3.5.4, p. 70).
bandwidth management	The service provider should ensure that all clients get a fair share of available bandwidth, but this aspect is not taken into account in our case study. Some execution environments offer better support than plain TCP/IP for resource control (see **accounting** and § 3.5.5, p. 71).
deployment	Although this possibility is not exploited in the case study, the service could deploy itself on several nodes (see **infrastructure dynamicity** and **parallelism**). Code mobility allows client to install mobile extensions on the service's platform (§ 9.1, p. 167).
infrastructure dynamicity	In (Tschudin, 1999) the dynamic deployment of a distributed service on all available platforms of a network is described.
load balancing	Not handled in the case study. Using mobility for this aspect (process migration) is an old idea, e.g. (Shehory et al., 1998).
memory management	The execution environment can provide low-level support (see **accounting** and § 3.5.5, p. 71).
parallelism	Not handled in the case study. Other works have shown that it is possible to parallelize a complex search procedure on available computing platforms for increased performance (Queloz, 1997).
prioritization	The execution environment can provide low-level support (see **accounting** and § 3.5.5, p. 71).
scripting	The service provider doesn't need to define a language or to program an interpreter, since the execution environment readily executes mobile extensions (Queloz and Villazón, 1999).

10.1.4 Remaining aspects

These aspects don't seem to be influenced by mobile code in our work, but we may have missed some nice opportunities to handle them in an easy way. We present them here just for the purpose of completeness.

Aspect	**Handled by**
auditing	Handled by simple logs in the case study.
encryption	Not handled in the case study, but may be necessary to hide passwords in the database. An encrypted connection (SSL) is usually used to protect the dialog with the Web clients.
historic	Not studied.
input media selection	Not studied.
internationalization	Not studied.
media synchronization	Not studied.
parameterization	The service is able to store preferences in the database, thus the user can access the service from anywhere using a Web browser, and he will retrieve his usual configuration (a benefit of the thin client model).
payment	Not studied (§ 3.5.5, p. 71).
persistence	In Meety, a relational database is used to store persistent data.
replication	Not studied.
storage layout	This aspect is encapsulated in entity and manager objects, according to the underlying database technology.
user interface look and feel	This aspect is not present in the service core, it is confined in the Web interface (e.g. Servlets) and thus easier to change. References between entities are represented in a portable format (e.g. long integers).
versioning	Not studied.

10.1.5 Discussion

Among the 38 non-functional aspects of our inventory (Chap. 2), only 13 don't seem to benefit from mobile code at all. For almost two thirds of them, the proposed approach offers one or several important simplifications, either because of the availability of code mobility or because of the features of a suitable execution environment. Moreover, there are benefits for both the service provider, and for the clients, hence our belief that the approach can have a very positive impact, if used for the implementation of large scale distributed applications.

Even if the possible improvements are very difficult to demonstrate empirically, because the primary impact is on non-functional aspects and not on primary functional aspects, and thus doesn't enable completely new "killer applications", we think that large amounts of efforts could be avoided with our approach. First coding efforts, because some aspects can be handled directly by the execution environment, hence coded only once, but also standardization efforts which frequently fail to deliver the expected results.

10.2 Concrete usage of Meety

- The service was first available in December 1999. Until today (end of April 2001), 135 people have requested a password.
- There are approximately 5 users, regularly coming back to organize new meetings.
- An average of 2 or 3 meetings are organized each month since december 1999.
- The service was announced one or two times to an estimated audience of a few hundreds to a couple of thousand potential users, within the University, and on a few scheduling mailing lists. It is also indexed by a couple of search engines but no other efforts were made to promote its use.

Clearly, these values are extremely modest, and the computational resources required are completely insignificant. It is difficult to explain why it is not used more often. Most feedback we got was rather positive, except the following:

- Someone in a company would have liked to use it but they didn't want to store any meeting information in a database outside of their organization.
- Someone would have preferred a french interface.
- Someone told us that he prefers to call participants on the telephone because it allows him to speak of other things (the communication is poorer with the service).
- Someone had problems with participants who didn't check their mail often enough.

On the other hand, our practical experience with the organization of real meetings using the system has shown that Meety is very convenient to find a suitable date for more than 10 people. Moreover, in several cases it was necessary to reschedule a meeting because of a change in the constraints (change in the availability of one participant, addition of a new participant). Finding a new possible date was very easy for the organizer and did not bother the participants since their answers had been stored in the system, and enough dates had been proposed. The fact that Meety sends invitations and confirmations automatically (and resends invitations to participants who fail to answer) relieves the organizer from doing tedious messaging. It makes our solution completely incommensurable with traditional ways to organize meetings using phones, fax or email.

Possible explanations of the low observed use are: (1) that we have not reached the right audience, since the efforts we made to promote the service were extremely limited; (2) that people are not yet used to such services, and don't understand the possible benefits; or that they don't have access to the Web, at the time they want to organize the meeting.

Apparently, the company that started TimeDance last year and which is now out of business met the same kind of problems, although they had certainly invested a lot of money and energy to promote their service.

Chapter summary

The first section on non-functional aspects shows how our case study validates the idea of an easier management for all these concerns and hence of services that are easier to build and to maintain in operation.

The second section reminds us that innovations are never guaranteed to succeed, but it doesn't discard our case study, since anyone that really needs a similar service (even in a completely different domain) will know the advantages of building it with our architecture and will be able to follow our general indications.

Part III

Perspectives and conclusions

In this last part of the thesis, we discuss some additional points, which need further study or that could not be treated in our research. Then we summarize the contributions of the dissertation, and the benefits of our approach.

Chapter 11

Future work

We have presented an architecture that exploits mobile code environments in order to implement services, which are easy to extend and to integrate. In this chapter, we discuss some difficulties that haven't been solved and we sketch possible solutions that must still be verified.

11.1 References between services

One problem that we have observed in our case study but that we haven't solved occurs when information that is referenced outside of a service must be deleted. Suppose that service A is built according to our architecture, and that service B references an instance I managed by A. Among the operations (use-cases) of A, one can be invoked by the owner of I to delete this information. D will denote this operation. Within A, there can be **integrity rules** that ensure that deleting I doesn't bring the system in an incoherent state, either by preventing the deletion by D or by updating or deleting further entities that reference I. Enforcing a complex set of integrity rules is not an easy task, but it becomes much more difficult when I is referenced within another service B, unknown to A. Additional conventions are necessary to keep coherence when D is executed.

This problem occurred with Meety because service A, which manages the address books and provides the mapping from email addresses to user IDs is separated from B, the service that manages other meeting informations. When the owner of an address book removes an entry I, by interacting with A, there may be informations in B that still depend on I, like the list of participants of a meeting, or information specific to A like the time zone of the corresponding user. Thus, I cannot simply be dropped from A's tables, even if there is no more address book referencing this user.

The classical solution of this problem, which is related to the aspects of **accuracy maintenance** and **memory management**, is to count references. But this mechanism doesn't tolerate that a single decrementation is forgotten, in a distributed context, where the counter is managed by A but no control on B is possible, this can be unrealistic. Better answers could be **leasing**, where counters are automatically decreased after a certain time, unless the lease has been renewed; or **market based allocation** where data is deallocated when no more sponsor is interested in it.

More work is needed to better evaluate the actual programming overhead required by these approaches. In our experiments with `MØ`, which offers a fair amount of low level support (presence of a sponsoring account for each piece of volatile memory, automatic deallocation by the platform), we found that the amount of work required to ensure proper sponsoring is not negligible, although terminating a service is greatly facilitated by this approach. However, not only volatile memory, but also persistent memory requires leasing mechanisms, and with our architecture and current database technology, this requires the explicit management of additional data structures (e.g. a timer for each entity in a table), as well as additional threads of control to check timer expiration.

We can speculate that this problem could well be handled at the level of database **managers**, by storing one additional expiration field for each entity, and running one thread to clear expired ones. An additional method to renew a lease would also be required. Last, each manager would be responsible to renew leases for "external" entities referenced by its own entities. This could require a fair amount of activity and additional storage because one timer is potentially required for each external entity. But, instead of a "delete" operation that blindly clears entities without taking into account external references, this would be an automatic mechanism for data deletion. Once an entity is deleted, further entities referenced in other tables and other services automatically become available for deallocation.

There would probably be an interesting role for mobile extensions in this context, given their ability to work in disconnected conditions. They could for instance be used to renew leases without incurring network traffic, provided that additional mechanisms to terminate an extension are available.

There is a strong similarity between the mechanism above and some functional parts of the service, which send notifications when a timer expires. Hence, the aspect of **Event notification** is also related and for some simple cases, the problem could be solved by propagating a signal from A to B when event D occurs. Being informed that the event has occurred, B can update its own information accordingly (delete invalid information, store a copy of I locally...), even without a fully-fledged leasing mechanism. Again, mobile extensions can play an important role, by handling the event at the source, and bringing the right amount of information, at a time that is suitable to B.

One final remark is that this section discusses only one kind of integrity rule, maybe the one that occurs most frequently, but there are several additional ones that may need more elaborate mechanisms than event notification and leasing. In any case, we think that there is an important investigation field in this direction.

11.2 Decorators for customized behavior

In the previous section, the operation D is executed by A whenever requested by the user. Although there may be some strong requirements in B to alter its behavior, we didn't consider this possibility. However, in several cases, we would have liked to allow that extensions or external services alter the behavior of the service's operations. For instance, change the output that is returned to the user. In the address book example above, a message could be displayed saying "Don't remove this person from your address book because it participates in one of your meetings".

The **decorator** design pattern of (Gamma et al., 1995) wraps an object around another one in order to alter its methods and to change its behavior. Detecting when an object receives a message, and being able to reprogram the event is also the idea of reflective architectures and meta-object programming. We think that this kind of techniques are necessary in order to allow extensions that modify the system function, or alter the interaction with the end user. Without them, extensions are still very useful, but their role is limited to interacting with the service using its software interface. They can be used to program new use-cases and new interactions with the end user, but they can't really modify existing ones. Schemes based on the invocation of methods before and after the event offer some control, notably because an exception can cancel the execution, but decorators are intrinsically able to perform more operations, like changing parameters, invoking the decorated method several times, etc.

In principle, it is mostly at the level of control objects that such techniques can be applied, since control objects are responsible to conduct the sequence of events and the instantiation of entity and interface objects.

It will be necessary to study whether decorators can substitute meta-object programming in those contexts where the latter is not available. One solution could be to use factories that can be configured in such a way that they return the decorators instead of the base objects. Obviously, this would require redefining parts of the architecture, especially the servlets that assume some of the responsibilities of control objects. It will also be necessary to restrict the scope of the decorators, in order to avoid security problems. Allowing that a given user decorates only his own meetings and personalizes his interactions with the system seems possible, but the situation can become tricky when several services are involved, without special trust relationships between them! Other problems are "stacking" of decorators, termination of a decoration, etc.

One good test case could be an extension that is able to guarantee an anteriority constraint between two meetings: while the end user is interacting with Meety using his Web browser, the extension checks that the date allocated to meeting i is before the date allocated to meeting j. The extension must be able to warn the user if the order is wrong or if there is no possibility left for the second meeting. It is necessary to receive event notifications when the user schedules a meeting, but also to display an information to the user. With the current implementation, the extension cannot warn the user within the flow of Web based interaction, but requires a dedicated client to open a new window on the user's workstation.

Meta-object programming or decorators can also be extremely useful because they potentially replace event notifications. Instead of explicitly defining observables and sending event notifications to observers, observers directly insert their code on the invocation path. Of course, there are consequences both from the point of view of security and of performance, but such alternatives can be worth studying since defining events is usually a time consuming task.

11.3 Extension repository

One effect of making the flow of control more explicit, as suggested in the previous section, is to reduce the role of Servlets and to increase the responsibilities of control objects and extensions. Servlets become a kind of communication channel between the Web user and the software effectively carrying on the operations. At some point, control objects and extensions could become undistinguishable software entities, that have sometimes been called "agents". These agents interact with the user by generating HTML pages, which are sent through the channel of Servlets. Answers from the user come back as HTTP requests, which lead to new pages, and so on. This is similar to the "wizards" that decompose a complex user interface in several simple steps.

In Chap. 4 we presented eAuctionHouse and Tabican. These two systems have shown the feasibility of this technique. An interesting future development would be to have a similar system for the management of agents, independent from a specific application. It should let end users manage, and interact with active extensions, and create new ones and parameterize them. Another side of the service should let extension developers enrich the set of available extensions, with suitable descriptions, code, discussions, etc.

The idea of a market for mobile agents, similar to the existing market for reusable components, has already been evoked several times, but it has not been achieved yet. In the case of Meety, we would have liked to find such a generally available service. We abandoned the idea of developing our own software for the management of extensions because of lack of time. But we are sure that showing our users how to interact with agents, using their Web browser, would have been a good feature to impress them.

11.4 Mobile code and non-functional aspects

The effect of mobile code on several non-functional aspects should also be studied. Because having the possibility to perform computations on both sides of the communication link potentially enables new solutions.

11.4.1 Caching

Caching implies duplication of data, and hence, the necessity to update or invalidate the replica when the original data is changed or deleted. This is frequently done by associating a timestamp to data, in order to record the last time it was modified. It could be interesting to study how event notifications for data changes can replace timestamps, in terms of ease of programming, scalability, performance, etc. For the service provider, event notification may be more interesting, e.g. require less memory or programming, when the aspect of event notification is already well handled by the service design. In this case, the ability of mobile extensions to filter events at the source or store them when the client is disconnected can be exploited. Mobile code may also offer more possibilities with respect to where the cache is located.

11.4.2 Coordination

In our case study, coordination was handled using Java's synchronisation mechanism at several well-chosen places in order to avoid race conditions. However, we have not studied this problem in depth, and we think that it is worth looking for new solutions enabled by mobile code. Because interacting entities can be co-located, and also because one entity that moves may replace several threads running on different machines, coordination problems may become easier to manage in this new context. At the same time, it is necessary to avoid that extensions, decorators, and all other kinds of mobile code obtain an exclusive access to critical resources and interfere with other extensions, or with the service's core functionality.

11.4.3 Exception handling

Like coordination, this aspect has not been treated very deeply in our architecture, but we think that mobile code enables original solutions. First, there is the example presented in § 9.4, p. 177 where the extension is able to move with an ordered sequence of events. An interesting observation, since it is usually very difficult to reconstruct the sequence of event that leads to a given state in a distributed setting. Second, we noticed that with our architecture it is possible to change the state of entity objects in a non-definitive way and thus, either to commit the change, or to leave the database unchanged if some exception occurs before the sequence is able to run to completion. It could be an interesting alternative to "transactional" systems that can also return to the last consistent state after a failure occurs. Hence, the ability to co-locate interacting entities, and to program the exception handling occuring on a remote host seem to offer interesting opportunities that would be worth studying in greater detail.

Chapter summary

We think that our work can be extended by several interesting studies of the aforementioned points. The meeting scheduling application, and the existing Meety software seem seem to be good starting points to conduct such studies in a real-world context.

Chapter 12

Summary of contributions

1. The first contribution of this dissertation is the extensive **catalog of non-functional aspects**. This inventory concretely depicts the complexity of large-scale software systems (DEDIS), in which most aspects are present. The large number of concerns that must be taken into account and the dense network of inter-dependencies somehow explain why software design is intellectually challenging. It makes evident the need for elaborate software engineering methods and tools, and for ways to handle the concerns separately.

2. We emphasize mobile code's most interesting feature: the **ability to encapsulate protocols**, which provides incomparable flexibility and allows integration with minimal collaboration. We contrast this approach with protocol based approaches where huge efforts are required to take into account the specific needs of all interested parties, to deploy and preserve the resulting "standards" and where ambiguities in specifications frequently lead to incompatible implementations. We provide several concrete examples of protocol encapsulation using mobile code throughout the dissertation.

3. We show that as a side-effect of protocol encapsulation, the ability to move code, and the availability of execution environments actually allow that a **significant fraction of non-functional aspects** are taken into account in better ways than with competing technologies for distributed systems. We infer that the motivation to use this technology will not be based on an hypothetic "killer application", because there is not a single aspect where mobile code excels and that cannot be handled with other techniques, but because it makes the work of the designers of DEDIS

easier, eventually leading to software with less defaults, and which is easier to adapt to changing needs and easier to integrate.

4. We give a detailed description of an **architecture for open and extensible services**. Unlike most existing work on mobility, which concentrated on the mobile parts of distributed applications, we show what is actually necessary to receive mobile entities. Although the architecture doesn't allow full reprogrammability of the service, we argue that such an approach offers important guarantees of evolution and accessibility of information and events. Such features are usually not present in desktop or client/server applications, leading to obsolete systems, or information that cannot be accessed although represented in an electronic form. Hence, we suggest that corporate IT departments change their point of view and consider offering reusable and personalizable services. Once there will be enough such services and sufficient security and continuity guarantees are offered, they can be used by third parties with only little collaboration.

5. As a case study, we implemented a **new Web-based service** for choosing a suitable date collaboratively. This is a useful tool, freely available for all Internet users. Since it is designed according to the thin client model, a user can access his meeting information from any computer connected to the Internet. This lets him easily introduce in the system the informations required to organize a meeting, and lets him retrieve them at any time. Such a system is a useful complement for all kinds of electronic and non-electronic personal calendars. We argue that for most end users, the ability to store information on the Web, instead of files on ones hard drive has many practical advantages (high availability, low administrative cost, etc.). Moreover, a service that can be extended using mobile code is a guarantee that it can survive technological changes and remain useful through programmability even when the user's needs evolve.

6. To implement our meeting scheduling service, using Java, we had to develop a new library of **classes for calendaring** and for the management of temporal intervals and domains. This package can be reused by other Java developers working on similar problems, since it has been released as open source software.

7. We also present in detail the **implementation of actual extensions** (mobile agents) within the Voyager environment. They illustrate well the advantage of protocol encapsulation, and of letting the client choose implementation details unilaterally. A set of high level conventions between

the service provider and its users are necessary, notably the descriptions of classes that can be accessed as service API. But there are many low-level tasks and details that the extension programmer needs to choose to actually deploy useful agents, and he may do so without collaboration with the service provider, and without going through heavy standardization processes. These extensions also enrich the rather sparse list of mobile agents effectively published and used.

There is no mobile code platform that satisfies all our requirements at the present time, but we don't think that we made unrealistic assumptions in this dissertation. It may take some time until an environment that offers the right combination of security, ease of use, performance and resource control becomes available, mostly because it represents a huge programming effort, but it should definitely be feasible. Then, the use of mobile code technology at the level of applications should become more frequent, given the potential advantages for end users, service providers and corporate IT departments, especially as an investment towards future integration. This day, the job of IT staff that often struggles to adapt software that is too stiff will hopefully be easier, and the experience of end users will improve, in front of systems that will be more robust and easier to personalize than today.

Bibliography

Ad Astra Engineering, Inc. (1998). Jumping beans white paper. Available on request on http://www.JumpingBeans.com/.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Baldi, M. and Picco, G. P. (1998). Evaluating the tradeoffs of mobile code design paradigms in network management applications. In Kemmerer, R. and Futatsugi, K., editors, *Proceedings of the 20th International Conference on Software Engineering (ICSE'97), Kyoto (Japan).*

Berry, P. M. (1992). The PCP: A predictive model for satisfying conflicting objectives in scheduling problems. *AI in Engineering*, 7:227–242.

Biberstein, O. (1997). *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva.

Binder, W., Hulaas, J., and Villazón, A. (2000). Resource control in J-SEAL2. Cahier du CUI 124, University of Geneva, Switzerland. ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.ps.

Booch, G., Jacobson, I., Rumbaugh, J., and Rumbaugh, J. (1998). *The Unified Modeling Language User Guide.* The Addison-Wesley Object Technology Series. Addison-Wesley Pub Co.

Brewington, B., Gray, R., Moizumi, K., Kotz, D., Cybenko, G., and Rus, D. (1999). Mobile agents in distributed information retrieval. In Klusch, M., editor, *Intelligent Information Agents.* Springer.

Burke, P. and Prosser, P. (1994). The distributed asynchronous scheduler. In Zweben, M. and Fox, M. S., editors, *Intelligent Scheduling*, chapter 11, pages 309–339. Morgan Kaufmann.

Campione, M., Walrath, K., Huml, A., and the Tutorial Team (1999). *The Java tutorial continued: the rest of the JDK*. Addison-Wesley, Reading, MA, USA. http://java.sun.com/docs/books/tutorial/.

Cesta, A., Collia, M., and D'Aloisi, D. (1998). Tailorable interactive agents for scheduling meetings. In Giunchiglia, F., editor, *Artificial Intelligence: Methodology, Systems, Applications*, volume 1480 of *Lecture Notes on Artificial Intelligence*. Springer.

Cesta, A. and D'Aloisi, D. (1999). Mixed-Initiative Issues in an Agent-Based Meeting Scheduler. *User Modeling and User-Adapted Interaction*, 9(1-2):45–78.

Chvátal, V. (1983). *Linear Programming*. W. H. Freeman and Company, New York.

Coleman, D. (1999). Web-based scheduling. online report http://www.collaborate.com/hot_tip/tip0799.html.

Committee on Information Technology Research in a Competitive World (2000). *Making IT Better: Expanding Information Technology Research to Meet Society's Needs*. National Academy Press, http://stills.nap.edu/html/making_IT_better/. ISBN 0-309-06991-2.

Conry, S. E., Meyer, R. A., and Lesser, V. R. (1988). Multistage negotiation in distributed planning. In Bond, A. H. and Gasser, L., editors, *Readings in Distributed Artificial Intelligence*, pages 367–384. Morgan Kaufman.

Dijkstra, E. W. and Scholten, C. S. (1990). *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.

Florio, S. (1998). Notes R5 calendar & scheduling. http://notes.net/today.nsf/9148b29c86ffdcd385256658007aaa0f/8894043cf3b42db8852566c50058d8d9?OpenDocument.

Freuder, E. C. and Wallace, R. J. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70.

Friha, L. (1998). *DISA: Distributed Interactive Scheduler using Abstractions*. PhD thesis, Université de Genève.

Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5).

Fünfrocken, S. (1998). Integrating java-based mobile agents into web servers under security concerns. In *31st Hawaii International Conference on System Sciences*, volume VII, Software Technology Track, pages 34–43.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

Garfinkel, R. S. and Nemhauser, G. L. (1972). *Integer Programming*. John Wiley & Sons, New York. Series in Decision and Control.

Glover, F. (1989). Tabu search: 1. *ORSA Journal on Computing*, 1(3):190–206.

Greppin, C. (2000). Réalisatoin d'un service Web de gestion de carnets d'adresses pour Meety. Mémoire de Licence, Université de Genève.

Grudin, J. and Palen, L. (1995). Why groupware succeeds: Discretion or mandate? In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, Electronic Meetings II, pages 263–278.

Hamadi, Y. (1999). *Processing of distributed constraint satisfaction problems*. PhD thesis, Université Montpellier II. http://magnum.lirmm.fr/~hamadi/Thesis/summary.html.

Harrison, C. G., Chess, D., and Kershenbaum, A. (1995). Mobile agents: Are they a good idea? Technical report, IBM Research Division, T. J. Watson Research Center, http://www.research.ibm.com/massdist/mobag.ps.

Hartvigsen, G., Johansen, D., Farsi, V., Farstad, W., Høgtun, B., and Knudsen, P. (1994). The StormCast API, specification of software interfaces in StormCast 2.1. Technical Report 94-19, Department of Computer Science, University of Tromsø, Norway. http://www.cs.uit.no/forskning/rapporter/Reports/9419.html.

Held, M. and Karp, R. M. (1970). The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162.

Hillier, F. S. and Lieberman, G. J. (1990). *Introduction to Operations Research*. McGraw–Hill, fifth edition.

Holland, J. H. (1992). *Adaption in Natural and Artificial Systems*. MIT Press.

Huai, Q. and Sandholm, T. (2000). Nomad: Mobile Agent System for an Internet-Based Auction House. *Internet Computing*, 4(2):80–86.

International Organization for Standardization (1988). *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times*. International Organization for Standardization, Geneva, Switzerland. http://www.iso.ch/markete/8601.pdf.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1995). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

Johansen, D. (1998). Mobile agent applicability. *Lecture Notes in Computer Science*, 1477:80–98.

Johansen, D., Lauvset, K. J., and Marzullo, K. (2000). An extensible software architecture for mobile components. Technical Report 2000-37, Department of Computer Science, University of Tromsø, Norway. http://www.cs.uit.no/forskning/rapporter/Reports/200037.html.

Kiczales, G. et al. (1997). Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming ECOOP'97*, Finland.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimisation by simulated annealing. *Science*, 220:671–680.

Kotz, D. and Mattern, F., editors (2000). *Agent Systems, Mobile Agents, and Applications*, number 1882 in Lecture Notes in Computer Science, Zurich, Switzerland. Springer.

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44.

Lai, C., Gong, L., Koved, L., Nadalin, A., and Schemers, R. (1999). User authentication and authorization in the java platform. In *Proceedings of the 15th Annual Computer Security Applications Conference*. http://java.sun.com/products/jaas/.

Lange, D. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.

Lea, D. (1997). Design for open systems in java. In Garlan, D. and Métayer, D. L., editors, *Proceedings COORDINATION'97*, LNCS 1282, pages 32–45, Berlin, Germany. Springer-Verlag.

Lugmayr, W. (1999). *Gypsy: A Component-Oriented Mobile Agent System*. PhD thesis, Technischen Universität Wien. http://www.infosys.tuwien.ac.at/Gypsy/download/docs/gypsy-diss.ps.gz.

Maes, P. (1994). Agents that Reduce Work and Information Overload. *CACM*, 37(7):30–40.

Milojicic, D. (1999). Trend wars. *IEEE Concurrency*, 7(3):80–90.

Minar, N., Gray, M., Roup, O., Krikorian, R., and Maes, P. (1999). Hive: Distributed agents for networking things. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA.

Mowbray, T. J. and Ruh, W. A. (1997). *Inside CORBA: Distributed Object Standards and Applications.* The Addison-Wesley object technology series. Addison-Wesley, Reading, MA, USA.

Muhugusa, M. (1997). *Distributed Services in a Messenger Environment: The Case of Distributed Shared-Memory.* PhD thesis, Computer Science Department, University of Geneva, Geneva, Switzerland.

Nadel, B. (1989). Constraint Satisfaction Algorithms. *Computational Intelligence*, 5(4):188–224.

Nesterov, Y. and Nemirovskii, A. (1994). *Interior-Point Polynomial Algorithms in Convex Programming*, volume 13 of *Siam Studies in Applied Mathematics.* Society for Industrial & Applied Mathematics.

ObjectSpace, Inc. (1999). *Voyager ORB 3.2 Developer Guide.* 14850 Quorum Drive, Suite 500, Dallas, TX 75240 USA. http://www.objectspace.com/.

Papaioannou, T. (2000). *On the Structuring of Distributed Systems: The Argument for Mobility.* PhD thesis, Loughborough University. http://www.luckyspin.org/Docs/Thesis-Book.zip.

Queloz, P.-A. (1997). Problem Solving in M0, Scheduling and resource allocation with VAD-heuristic. Cahier du CUI 111, University of Geneva, Switzerland. http://cui.unige.ch/~queloz/papers/m0_vad.ps.gz.

Queloz, P.-A. and Pellegrini, C. (1999). Foreign event handlers to maintain information consistency and system adequacy. In *Workshop on Mobile Agents in the Context of Competition and Cooperation, Autonomous Agents Conference, Seattle.*

Queloz, P.-A. and Villazón, A. (1999). Composition of services with mobile code. In *First International Symposium on Agent Systems and Applications (ASA'99) and Third International Symposium on Mobile Agents (MA'99) ASA/MA'99*, Palm Springs, California, USA.

Rowstron, A. (1999). Mobile Co-ordination: Proving Fault Tolerance in Tuple Space Based Coordination Languages. In Ciancarini, P. and Wolf, A. L., editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 of *Lecture Notes in Computer Science*, pages 196–210, Amsterdam, Netherland. Springer-Verlag, Berlin.

Sen, S. and Durfee, E. H. (1995). Unsupervised surrogate agents and search bias change in flexible distributed scheduling. In Lesser, V., editor, *Proceedings of the First International Conference on Multi-Agent Systems*, pages 336–343, San Francisco, CA. MIT Press.

Sen, S. and Durfee, E. H. (1996). A contracting model for flexible distributed scheduling. *Annals of Operations Research*, 65:195–222.

Sen, S. and Durfee, E. H. (1998). A formal study of distributed meeting scheduling. Group Decision and Negotiation.

Sen, S., Haynes, T., and Arora, N. (1997). Satisfying user preferences while negotiating meetings. *International Journal of Human-Computer Studies*, 47(3):407–427.

Shehory, O., Sycara, K., Chalasani, P., and Jha, S. (1998). Agent cloning: An approach to agent mobility and resource allocation. *IEEE Communications Magazine*, pages 58–67.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition.

Sudmann, N. P. and Johansen, D. (2000). Adding mobility to non-mobile web robots. In *ICDCS'00 Workshop on Knowledge Discovery and Data Mining in the World-Wide Web.* http://www.cs.uit.no/forskning/rapporter/Reports/200036.ps.

Sun Microsystems (1997). *JavaBeans.* http://java.sun.com/beans.

Sun Microsystems Inc. (1999). *Jini Connection Technology.* Sun Microsystems Inc., http://www.sun.com/jini.

Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., and Jeffers, R. (2000). Strong mobility and fine-grained resource control in NOMADS. In (Kotz and Mattern, 2000), pages 2–15.

Tennenhouse, D. L. et al. (1997). A survey of active network research. *IEEE Communications Magazine*, pages 80–86.

Tiemann, M. et al. (April 1997). Information architecture volume III guidance. Technical report, U.S. Department of Energy. http://cio.doe.gov/iap/documents/vol3_guidance/volume3.htm.

Tripathi, A., Ahmed, T., Kakani, V., and Jaman, S. (2000). Distributed collaborations using network mobile agents. In (Kotz and Mattern, 2000), pages 126–137.

Tschudin, C. (1999). A self-deploying election service for active networks. In *Coordination'99*, Amsterdam, Holland.

Tschudin, C. F. (1993). *On the structuring of computer communications*. PhD thesis, University of Geneva, Switzerland.

Tschudin, C. F. (1994). An introduction to the M0 messenger language. Technical Report Cahier du Centre Universitaire d'Informatique 86, University of Geneva, Switzerland.

Tschudin, C. F. (1997a). The messenger environment M0 – A condensed description. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet (MOS'96)*, volume 1222 of *LNCS*, pages 149–156. Springer-Verlag, Berlin, Germany.

Tschudin, C. F. (1997b). Open resource allocation for mobile code. In *First International Workshop on Mobile Agents, MA'97 Berlin*.

Van Hentenryck, P., Michel, L., and Deville, Y. (1997). *Numerica: A Modeling Language for Global Optimization*. MIT Press, Cambridge, Mass.

Van Hentenryck, P. and Saraswat, V., editors (1995). *Principles and Practice of Constraint Programming*. MIT Press, Cambridge, MA.

Vigna, G. (1998). *Mobile Code Technologies, Paradigms and Applications*. PhD thesis, Politecnico di Milano. http://www.cs.ucsb.edu/~vigna/pub/vigna_PhDThesis97.ps.gz.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1994). A note on distributed computing. Technical report, Sun Microsystems Laboratories, Inc., http://www.sunlabs.com/techrep/1994/abstract-29.html.

Yamamoto, G. and Nakamura, Y. (1999). Architecture and performance evaluation of a massive multi-agent system. In Etzioni, O., Müller, J. P., and Bradshaw, J. M., editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 319–325, New York. ACM Press.

Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *12th International Conference on Distributed Computing Systems*, pages 614–623, Washington, D.C., USA. IEEE Computer Society Press.

Zweben, M. and Fox, M. S., editors (1994). *Intelligent Scheduling*. Morgan Kaufmann.

Index