CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications

Licia Capra, Wolfgang Emmerich and Cecilia Mascolo

Abstract— **Mobile devices, such as mobile phones and personal digital assistants, have gained wide-spread popularity. These devices will increasingly be networked, thus enabling the construction of distributed applications that have to adapt to changes in context, such as variations in network bandwidth, battery power, connectivity, reachability of services and hosts, and so on. In this paper we describe CARISMA, a mobile computing middleware which exploits the principle of reflection to enhance the construction of adaptive and context-aware mobile applications. The middleware provides software engineers with primitives to describe how context changes should be handled using policies. These policies may conflict. We classify the different types of conflicts that may arise in mobile computing and argue that conflicts cannot be resolved statically at the time applications are designed, but, rather, need to be resolved at execution time. We demonstrate a method by which policy conflicts can be handled; this method uses a micro-economic approach that relies on a particular type of sealed-bid auction. We describe how this method is implemented in the CARISMA middleware architecture, and sketch a distributed context-aware application for mobile devices to illustrate how the method works in practise. We show, by way of a systematic performance evaluation, that conflict resolution does not imply undue overheads, before comparing our research to related work and concluding the paper.**

Keywords— **Middleware, mobile computing, reflection, context-awareness, conflict resolution, game theory, quality of service.**

I. Introduction

MOBILE computing devices, such as palmtop computers, mobile phones, personal digital assistants (PDA) and digital cameras have gained wide-spread popularity. These devices will increasingly be networked and software development kits are available that can be used by third parties to develop applications [1].

Even though devices and networking capabilities are becoming increasingly powerful, the design of mobile applications will continue to be constrained by physical limitations. Mobile devices will continue to be battery-dependent and users are likely to be reluctant to carry heavy-weight devices. Wide-area networking capabilities will continue to be based on communication with basestations, with fluctuations in bandwidth depending on physical location. In order to provide acceptable quality of service to their users, and consequently improve user satisfaction of the system, applications have to be *context-aware* [2], and able to adapt to context changes, such as variations in network bandwidth, exhaustion of battery power or reachability of services on other devices. This would require application engineers, for example, to periodically query heterogeneous physical sensors, in order to get updated context information, to detect context configurations of interest to the application, and adapt accordingly; however, doing so would be extremely tedious and error-prone.

In order to ease the development of context-aware applications, middleware layered between the network operating system and the application have to provide application engineers with powerful abstractions and mechanisms that relieve them from dealing with low-level details. For example, applications must be able to specify, in a uniform way, which resources they are interested into, and which behaviours to adopt in particular contexts. The middleware, then, maintains, on behalf of the applications, updated context information, detects changes of interest to the application, and reacts accordingly.

In the past decade, the development of distributed applications for wired systems has been greatly enhanced by middleware systems that succeeded in facilitating the communication between distributed components. Traditional middleware systems (e.g., CORBA, Java/RMI, MQSeries) provide application engineers with communication abstractions that relieve them from dealing with, for example, the location of distributed components, network failures and hardware heterogeneity (e.g., for marshaling/unmarshaling of parameters). These middleware are based on the principle of transparency [3][4]: implementation details are hidden from both users and application designers and are encapsulated inside the middleware itself, so that the distributed system appears to application developers as a single integrated computing facility.

Although having proved successful in supporting the construction of traditional distributed systems, we argue that transparency cannot be used as the guiding principle to develop the new abstractions and mechanisms needed by mobile computing middleware to foster the development of context-aware applications. By providing transparency, middleware must take decisions on behalf of the application; this is inevitably done using built-in mechanisms and policies that cater for the common case rather than the high levels of dynamicity and heterogeneity intrinsic in mobile environments. Applications, instead, may have valuable information that could enable the middleware to execute more efficiently, in different contexts.

We argue that *reflection* [5] offers significant advantages for building mobile computing middleware. A reflective system may modify its own behaviour by means of inspection (i.e., the internal behaviour of the system is exposed)

L. Capra is with the Department of Computer Science, University College London, London, UK. E-mail: l.capra@cs.ucl.ac.uk.

W. Emmerich is with the Department of Computer Science, University College London, London, UK. E-mail: w.emmerich@cs.ucl.ac.uk.

C. Mascolo is with the Department of Computer Science, University College London, London, UK. E-mail: c.mascolo@cs.ucl.ac.uk.

and/or adaptation. (i.e., the internal behaviour of a system can be dynamically changed). For example, applications may dynamically alter the set of resources that middleware monitors on their behalf, the context configurations they are interested into, and the behaviours they want to adhere to. However, while doing so, applications may introduce ambiguities, contradictions, and other logical inconsistencies. For example, contradictory behaviours may be requested by the same application to react to a particular context change, or cooperating applications may not agree on a common behaviour to be applied. We refer to these inconsistencies as *conflicts*.

The novel contribution of this paper is the design, formalisation and evaluation of new abstractions and mechanisms that, embedded in a mobile computing middleware software layer, facilitate the development of context-aware applications. In particular, we exploit the principle of reflection to achieve dynamic adaptation to context changes: we offer applications an abstraction of the middleware as a dynamically customisable service provider, where customisation takes place by means of metadata, which encode middleware behaviour to answer application service requests in various contexts. Through reflection, the meta-information can be changed, and therefore the middleware behaviour tuned, with the risk, however, of incurring conflicts. We have designed a microeconomic approach to *conflict resolution* that relies on a particular type of sealed-bid auction. Our approach treats a distributed mobile system as an 'economy' where applications compete to have the middleware deliver the quality-of-service they desire. The mobile computing middleware plays the role of an auctioneer, collecting bids from applications and delivering services with the QoS requested by the successful one. We show why our auctioning mechanism is particularly useful in a mobile setting and that it achieves fair conflict resolution.

The remainder of the paper is structured as follows: Section II describes our reflective middleware model; Section III introduces the issue of conflicts in a mobile setting, provides a classification of the types of conflicts we deal with, and illustrates them using an example of context-aware application. In Section IV we formalise the microeconomic mechanism we propose to solve these conflicts, and illustrate a comprehensive example, based on the application described before, to clarify how the model works in practise. Section V evaluates our model in terms of usability and performance; Section VI compares our approach to related work, and finally Section VII concludes the paper and identifies possible future work.

II. The Reflective Model

Mobile applications execute in an extremely dynamic context: location changes all the time while moving around with our portable device, and so the services and devices in reach; local resource availability varies quickly as well, such as memory availability, bandwidth, and battery power. In order to provide reasonable quality-of-service to their users, applications have to be context-aware.

We argue that *reflection* [5] is a powerful means to build mobile computing middleware that supports the development of context-aware applications. The key to the approach is to make some aspects of the internal representation of the middleware explicit, and hence accessible from the application, through a process called *reification*. Applications are then allowed to dynamically inspect middleware behaviour (introspection), and also to dynamically change it (adaptation), by means of a meta-interface that enables run-time modification of the internal representation previously made explicit. The process where some aspects of the system are altered or overridden is called *absorption*. The whole process is depicted in Figure 1(a).

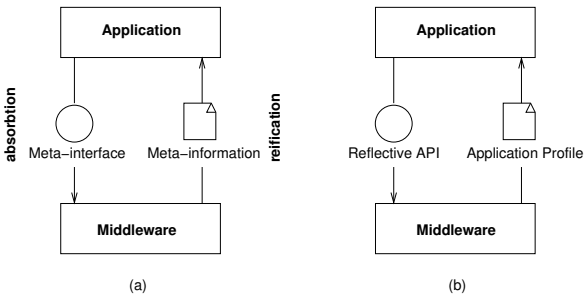

Fig. 1. The Reflective Process.

CARISMA, a project carried out at University College London, is a middleware model that exploits reflection to enable context-aware interactions between mobile applications. In our model [6], the middleware is in charge of maintaining a valid representation of the execution context, by directly interacting with the underlying network operating system. By context, we mean everything that can influence the behaviour of an application, from resources within the device, such as memory, battery power, screen size and processing power, to resources outside the physical device, such as bandwidth, network connection, location and other hosts within reach, to application-defined resources, such as user activity and mood.

Applications may require some services to be delivered in different ways (using different policies) when requested in different context. For example, a messaging application may wish to send messages in plain when bandwidth is high, while exchanging compressed messages when bandwidth is low.

To enhance the development of context-aware applications, CARISMA provides application engineers with an abstraction of the middleware as a customisable service provider. In particular, the behaviour of the middleware with respect to a specific application is described as a set of associations between the *services* that the middleware customises, the *policies* that can be applied to deliver the services, and the *context configurations* that must hold in order for a policy to be applied. In the example above, an association is defined between the 'messaging' service, the 'plain message' policy, and a context where the resource 'bandwidth' is high, and another one between the same 'messaging' service, the 'compressed message' policy, and a context where 'bandwidth' is low. The behaviour of the middleware with respect to a particular application is rei-

$$
\begin{aligned}
\mathit{serviceList} &::= \mathit{service}\ \mathit{serviceList} \mid \varepsilon \\
\mathit{service} &::= \mathit{sname}\ \mathit{policyList} \\
\mathit{policyList} &::= \mathit{policy}\ \mathit{policyList} \mid \mathit{policy} \\
\mathit{policy} &::= \mathit{pname}\ \mathit{contextList} \\
\mathit{contextList} &::= \mathit{context}\ \mathit{contextList} \mid \mathit{context} \\
\mathit{context} &::= \mathit{resourceList} \\
\mathit{resourceList} &::= \mathit{resource}\ \mathit{resourceList} \mid \varepsilon \\
\mathit{resource} &::= \mathit{rname}\ \mathit{oname}\ \mathit{valueList} \\
\mathit{valueList} &::= \mathit{value}\ \mathit{valueList} \mid \varepsilon
\end{aligned}
$$

Fig. 2. Application Profile Abstract Syntax. $\mathit{sname} \in \mathrm{S}$, $\mathit{pname} \in \mathrm{P}$, $\mathit{rname} \in \mathrm{R}$, being $\mathrm{S}, \mathrm{P}, \mathrm{R} \subset \Sigma^*$, respectively, the sets of all valid service/policy/resource names over our alphabet Σ. $\mathit{value} \in \mathrm{V}$, being V the set of all possible values of resources in R (e.g., IP addresses for hosts in reach, etc.); $\mathit{oname} \in \mathrm{O}$, being O the set of all valid operator names that can be applied to values of monitorable resources (e.g., *equals*, *lessThan*).

```
messagingService
    plainMsg
        bandwidth > 40%
    compressedMsg
        bandwidth < 40%
```

Fig. 3. Customisation of the Messaging Service.

fied in what we call an *application profile*, as shown in Figure 1(b)[1]. Figures 2 and 3 show respectively the profile abstract syntax and an example of a customised service encoded using this syntax.

Profiles are passed down to the middleware; each time a service is invoked, the middleware consults the profile of the application that requests it, queries the status of the resources of interest to the application itself, as declared in the profile, and determines which policy can be applied in the current context, thus relieving the application from performing these steps. Our model assumes that the behaviour of the middleware with respect to a particular service is determined, at any time, by one and only one policy, that is, a service cannot be delivered using a combination of different policies. More policies can logically be combined; for example, the messaging service can be provided with a 'compressed message' policy, that is a logical combination of two separate policies, 'compress' and 'send'. However, we regard the combined policy as a new one, and in the profile we will refer to this new policy, not to the sequential execution of two distinct policies.

As both the user needs and the context change quite frequently (e.g., due to movement of the device to a different location), we cannot expect application designers to foresee all possible configurations. Through a reflective API (Figure 1(b)), applications can dynamically inspect the content of their profile (i.e., the current configuration), and alter it by adding, deleting and updating the associations

[1]Our reflective middleware model assumes a single user for each mobile device, though there may be many applications running simultaneously on that device, hence, in our model, on the same middleware instance (this assumption is reasonable for portable devices, such as PDAs and mobile phones).

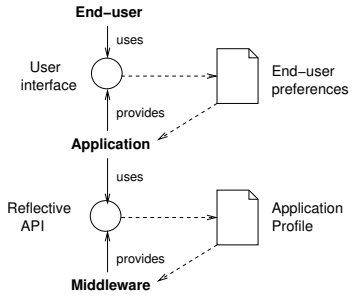

Fig. 4. Roles and Responsibilities in the Reflective Process.

previously encoded. As the behaviour of the middleware is dictated by the associations encoded in the application profiles, changing this information means dynamically affecting middleware behaviour (i.e., re-configuration of the system). If we consider once again the messaging example, an application may add an association to its profile requesting the execution of the messaging service using an 'encrypted message' policy, in order to achieve privacy of information, when both battery and bandwidth availability are high.

A default profile exists for every application, where each service that the middleware delivers (to that application) is associated to exactly one policy, regardless of context. It is up to the application to decide whether and when to exploit the power of reflection to alter the information here encoded, that is, to customise middleware behaviour in order to achieve better quality of service.

So far we have focused our discussion on the interaction between middleware and applications, leaving the end-users of the system behind the scene. As Figure 4 illustrates, the middleware provides applications a reflective API (i.e., meta-interface) that they can exploit to inspect and alter middleware behaviour, as encoded in application profiles. The target users of our middleware model are therefore application developers. In customising middleware behaviour, however, end-user requirements and expectations must be taken into consideration. We therefore expect applications built on top of CARISMA to provide end-users with a user interface through which end-user preferences can be captured, and used by applications to encode profiles.

In this paper, we are mainly concerned with the interaction between middleware and applications, thus leaving the issue of gathering user preferences and synthesising them in application profiles for future work; however, we will provide in Section V-B some insights on the complexity of doing so, and on the amount of work required from the user to teach the system to behave according to his/her own expectations.

III. Dealing with Conflicts

The model presented above allows applications to control the behaviour of the middleware based on current user needs and context. This is achieved by means of application profiles that can be dynamically changed through a reflective API. Although a middleware based on this model

supports the development of context-aware applications, it also opens the door to conflicts. In our model, a conflict exists when different policies can be used in the same context to deliver a service, so that the middleware does not know which one to apply (note that we made the assumption that a service can be delivered using only one policy at a time). Reflection gives applications the 'intelligence' that transparency takes away in traditional middleware systems. Applications, however, may not be smart enough to cope with the new power, and may produce profiles that lead to conflicts. In particular, when setting up application profiles, the following two basic kinds of conflicts may be created.

- **Intra-profile conflict**: a conflict exists inside the profile of an application running on a particular device. This class identifies conflicts that are *local* to a middleware instance.
- **Inter-profile conflict**: a conflict exists between the profiles of applications running on different devices. This class identifies conflicts that are *distributed* among various middleware instances. A particular case of inter-profile conflict happens when applications run on the same device (i.e., on the same middleware instance); we refer to this situation as an *N-on-1* (i.e., N applications on 1 device) *conflict*.

In order to understand how these types of conflicts arise, we sketch a conference application that is representative of the class of context-aware mobile applications that would benefit from our reflective middleware model; we then discuss the requirements that a conflict resolution mechanism must meet, before presenting the details of the mechanism we have designed. At this stage, we are not interested in implementation details (in particular, in the language used to encode profiles); we therefore use the abstract syntax illustrated before to discuss the following examples.

A. Conference Application

Let us imagine a researcher Alice travelling to a conference with her own PDA. When arriving at the conference location, she is provided with a Conference Application to be installed on her portable device that, based on a wireless network infrastructure, allows attendees to access the proceedings electronically, browse through the technical and social programme, select the talks they wish to attend and be alerted of the selected ones 10 minutes before they start, and exchange messages with other attendees. These services may have to be delivered in different ways when requested in different contexts, in order to meet the user's needs. Let us consider, in particular, the talk reminder service and the messaging service; through our reflective middleware model, Alice's preferences can be taken into account and used to generate the following associations.

Reminder of the next talk. The reminder functionality of the system can capture user attention through one of the following policies: `soundAlert`, particularly useful to capture user attention in noisy and open air places; `vibraAlert`, to capture user attention without disturbing anybody else (e.g., while attending a talk); and `silentAlert`, to remind the user of the next talk through a blinking message, for example, while the user is actively using the portable device. Figure 5 shows an example of Alice's associations for the `talkReminder` service.

```
talkReminder
    soundAlert
        location = outdoor
    vibraAlert
        location = conferenceRoom
    silentAlert
        userFocus = on
```

Fig. 5. Example of Local (Intra-profile) Conflict.

The `talkReminder` is an example of local service, as it does not require the cooperation of any other party. Let us consider, for example, the encoding shown in Figure 5, and let us assume that the service is requested when Alice is attending a talk (i.e., `location = conferenceRoom`), and using her PDA to take notes at the same time (i.e., `userFocus = on`). The middleware checks which policy should be applied and determines that more than one policy suits the current context (i.e., `vibraAlert` and `silentAlert`). As we made the assumption that each service is delivered using one and only one policy at a time, the middleware is unable to choose which of the context-suitable policies to apply[2]. This is an example of *intra-profile conflict*.

Exchange of messages. Attendees can exchange messages using any of the following policies: `charMsg`, that delivers one character at a time, `plainMsg`, to exchange messages in plain, `compressedMsg` to exchange compressed messages, and `encryptedMsg` to send encrypted messages.

The `messagingService` is an example of peer-to-peer service, where any number of peers may participate in the delivery of the service. In order for the service to be delivered, all the communicating peers have to agree on a common policy to be applied. Let us assume, for example, that Alice, Bob and Claire are willing to exchange messages; let us also assume that their profiles are the ones illustrated in Figure 6. Note that no context information is associated to the `plainMsg` policy of Bob's profile: this means that this policy is always enabled, regardless of current context. At any time, users may change their preferences through the user interface that the conference application provides; the application, in turn, dynamically updates the meta-information encoded in their profiles, in order to take the new preferences into account.

```
% Alice                 % Bob                 % Claire
messagingService        messagingService      messagingService
  plainMsg                plainMsg              plainMsg
    battery < 40%                               bandwidth > 50%
  encryptedMsg                                  compressedMsg
    battery > 40%                               bandwidth < 50%
```

Fig. 6. Example of Distributed (Inter-profile) Conflict.

If the `messagingService` is requested when battery

[2]Note that, by removing this assumption, we do not avoid the issue of conflicts, we just need to formulate it under different terms. In particular, conflicts would appear as different *sets* of policies enabled at the same time; in this case, we should consider whether the order in which policies appear in a profile is relevant, or whether their execution is commutative.

availability is below 40% on Alice's PDA, and Claire's bandwidth is greater than 50%, they all agree on the `plainMsg` policy to be applied; but what if Alice's battery is greater than 40%, or if Claire's bandwidth is lower than 50%? This is an example of *inter-profile conflict*.

B. Requirements

Whenever a service that incorporates a conflict, either intra- or inter- profile, is requested, a conflict resolution mechanism has to be run to solve the conflict and find out which policy to use to deliver the service, otherwise applications cannot execute. In designing such mechanism, the following requirements have to be considered.

Dynamicity. Neither intra- nor inter- profile conflicts can be detected and resolved statically, that is, at the time the profile is written by the application and passed down to the middleware. In case of intra-profile conflict, a possible static approach would require us to check whether there is any intersection between the different contexts of the policies associated to each service. Due to the complex nature of context (the number of monitored resources may be large), a static conflict analysis would produce an explosion in the context information that must be checked, and would require a consumption of resources (especially in terms of battery, memory and processing power) that portable devices cannot bear. Providing the conflict resolution as an external service on a powerful machine that is contacted on-demand is not feasible either, as this would require persistent connectivity that in mobile settings cannot be taken for granted. As for inter-profile conflicts, the situation is even worse; mobile devices connect opportunistically and sporadically. We cannot foresee which devices are going to be encountered and, even so, we cannot assume that all of them will be connected and in reach at the time a profile is modified; this means that the middleware cannot statically check whether the new configuration is conflict-free. Even assuming that this distributed check could be statically performed, it would not be worth the effort, as we would find many more potential conflicts than what we would actually need, as conflicts manifest themselves only with respect to the particular context in which the service is requested, and the profiles of the participating peers. As a consequence, a dynamic solution is needed: conflicts may exist inside or among profiles, but both applications and middleware can live with these conflicts until a service which incorporates a conflict is invoked.

Simplicity. The conflict resolution mechanism must be simple in the sense that it must not consume resources that are already scarce on a mobile device. Only a low computation and communication overhead should be imposed, even if this may occasionally prevent from an optimal solution to the conflict to be found.

Customisation. Middleware cannot choose how to solve conflicts independently of the applications that requested the conflicting service, as only the applications know how much they value the execution of the various policies. On one hand, we do not want applications to be questioned each time a conflict is detected, that is, middleware should be in charge of carrying out the conflict resolution process in an automatic way as much as possible. On the other hand, it must be possible for the applications to customise the conflict resolution mechanism, thus influencing which policy is chosen and applied, and which others are discarded.

In the following section, we formally describe a conflict resolution mechanism that meets these requirements.

IV. Microeconomic Mechanism

When applications participating in the delivery of a service cannot agree on which policy must be applied, a *dynamic* conflict resolution scheme is necessary to resolve the dispute. The conflict resolution mechanism we propose is based on microeconomic techniques [7]. The motivating idea is that a mobile distributed system can be seen as an *economy*, where a set of *consumers* must make a collective choice over a set of alternative *goods*. Goods represent the various policies that can be used to deliver a service (not the resources needed to apply a policy); for example, policies 'plainMsg', 'encryptedMsg' and 'compressedMsg' are the goods associated to service 'messagingService'. Consumers are applications seeking to achieve their own goals, that is, to have the middleware delivering a service using the policy that provides the best quality of service, according to application-specific preferences.

Simple schemes include, for example, priority assignment or per capita distribution. However, those do not suit situations where participation in exchange of goods is voluntary on the part of all parties (i.e., the applications), so that action requires a consensus and mutual perception of benefit. A better scheme would use an *auction protocol*. Auctions allow parties to make decisions independently, on the basis of private state, revealing only offers and acceptance of the offers made by others. Applications may vary greatly in their preferences and decision processes. An auction permits greater degrees of heterogeneity than simpler schemes.

The question we have to answer next is which auction protocol to use. This is known in microeconomic theory as a mechanism design problem [8]. A *protocol*, or mechanism, consists of a set of rules that govern interactions, by which agents (i.e., our applications) will come to an agreement. It constraints the deals that can be made, as well as the offers that are allowed. We argue that the auction protocol we have designed [9] can be successfully applied in a mobile setting, where the requirements listed in Section III-B must be satisfied.

A. The Protocol

The rules of our auction are very simple: given a setting with N agents that must make a collective choice from a set of P possible alternatives, each agent submits a single sealed bid for each element in P. The auctioneer collects the bids and selects the alternative in P that maximises social welfare, that is, the alternative with the highest sum of bids received. Each agent then pays the

auctioneer an amount of money that is proportional to the bid they placed on the winning alternative.

In our scenario, the role of the auctioneer is played by the middleware, which we assume is a trusted entity whose code and behaviour cannot be interfered with. Applications are the agents, and the good they are competing for is the execution of the policy they value most, among a set of alternatives that correspond to the policies that can be applied in a particular context to deliver a service. As previously said, the aim of the middleware is not to select the policy that received the highest bid (i.e., the one that maximises the selling price), but, rather, the policy that satisfies the largest number of applications involved in the conflict. In our scenarios, in fact, applications are participating in the delivery of the same service, rather than competing for it (i.e., the service will be delivered to all of them, not only to one or some of them). In these collaborative, or at least compromise, scenarios, a solution that satisfies the total benefit of all the applications is preferred to one that maximises the revenue of a single one.

Our auction has been inspired by traditional sealed bid auctions (e.g., first-price and second-price sealed bid auction [10]). Unlike ascending bid auctions, such as the standard English auction [11], where the auctioneer, adopting a possibly long iterative process, continuously raises the price of the good until only one bidder is willing to meet the price called, sealed bid auctions consist of a one-step bid that cuts down the computation and communication costs when the auction is distributed over space and time, as in our mobile setting. This meets our requirement of *simplicity*. We will show in Section IV-B how customisation is met by our auctioning mechanism.

In the following, we formalise the steps of our auctioning mechanism. We do not discuss here how coordination among different middleware instances takes place; details about the algorithm that implements this coordination can be found in Section V-A. To avoid confusion between an application (which may exist on different devices) and an application instance (which runs on a particular device), we will identify an application instance and the device it is executing on as a 'peer'. Peers are partners in the communication process. We call PEER the set of all possible peers. Under these assumptions, the auctioning process can be formalised as follows.

Initialisation. As part of an initialisation process, for every peer $peer_i$, $i \in [1, N]$, a utility function $u_i : \mathrm{P} \rightarrow \mathbb{R}^+$ that represents the user's goals (e.g., minimisation of consumption of resources, maximisation of quality of service, etc.) can be determined. Peers use their utility function to specify how much they value the use of a policy $p_j \in \mathrm{P}$ during an auction, that is, $u_i(p_j) = u_{i,j}$. Each peer is also assigned a quota q_i by the middleware. The quota q_i represents the maximum amount of money that $peer_i$ can bid during a bidding process, that is, the bid placed by peer $peer_i$ on policy p_j is a number $b_{i,j} = \min\{u_{i,j},\ q_i\}$.

Service Request. Whenever an application requires the middleware to execute a service, a command like the one illustrated below is issued:

$$\begin{aligned} command &::= sname\ peerList \\ peerList &::= peer\ peerList \mid peer \end{aligned}$$

being $sname \in \mathrm{S}$ the name of the requested service, and $peerList$ the set of peers involved in the service execution.

Assuming that service $sname$ requires the cooperation of $n \leq N$ peers, each peer (or, better, the middleware instance operating on the device of the peer) computes P_i as the set of policies that the above running application instance A_i has associated to service $sname$ in its profile, and that can be applied in the current context (i.e., according to current resource availability). More formally, P_i can be defined as follows:

$$P_i = \mathcal{F}[\![serv(sname, peer_i)]\!]_{\mathcal{E}nv(peer_i)}$$

$\mathcal{F}$ being the semantic function defined in Figure 23 in the Appendix; $serv : \mathrm{S} \times \mathrm{PEER} \rightarrow service$ a function that, given a service name and a peer, returns the corresponding service specification, and $\mathcal{E}nv : \mathrm{PEER} \rightarrow \mathrm{E}$ a function that computes the current execution environment of a peer.

Computation of the Solution Set. Middleware instances then cooperate to compute the *solution set* P^*, that is, the set of policies that all peers involved in the execution of the service have agreed upon:

$$P^* = \mathcal{I}[\![sname]\!]_{\{peer_1 \ldots peer_n\}}$$

$\mathcal{I}$ being the semantic function described in Figure 22 in the Appendix.

If the cardinality of P^* is zero, that is, the solution set is empty, a conflict exists that cannot be solved, as peers do not agree on a common policy to be applied; the conflict resolution process is terminated with a failure and peers are notified. If the cardinality is exactly 1, there is an agreement on the policy to apply (i.e., there is no conflict). Finally, if the cardinality is greater than 1, there is a conflict that can be resolved using one of the policies in P^*. In this case, the auctioning process proceeds as below, to decide which of these policies should be applied.

Computation of Bids. For every peer $peer_i$ participating in the communication process, and for every agreed policy $p_j \in P^*$, $j \in [1, m]$, a bid $b_{i,j}$ is computed, based on the peer utility function u_i and quota q_i. Unlike 'human' auctions, we make the assumption that all peers participating in a bidding process bid a price, that is, they cannot refuse to bid. Middleware instances of bidding peers exchange the bids they have received, ending up with a merged set of tuples $\mathcal{B}^*$ specifying how much each peer values the use of each agreed policy:

$$B^* = \mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peer_1, \ldots, peer_n\}}$$

$\mathcal{B}$ being the semantic function shown in Figure 24 in the Appendix.

Election of the Winner. From the set B^*, middleware instances participating in the conflict resolution process select the winning policy $p_{\bar{\jmath}}$ as the one with the highest sum of the bids placed:

$$p_{\bar{\jmath}} = \mathcal{W}[\![B^*]\!]$$

$\mathcal{W}$ being the semantic function defined in Figure 25 in the Appendix; as shown there, each peer also pays an amount of money that is proportional to the 'added' benefit obtained by applying the winning policy over the other peers. To understand how the payment is split, let us consider three peers x, y and z, who bid $b_x < b_y < b_z$ respectively on a winning policy p. Applying p gives an equal benefit of b_x to each peer; moreover, y and z share an added benefit of $b_y - b_x$ over x, and z enjoys an extra benefit equal to $b_z - b_y$ over both x and y. Our payment scheme demands that x, y and z pay 0, $(b_y - b_x)/2$, and $(b_y - b_x)/2 + (b_z - b_y)/1$ respectively. Note that, if the winning policy is the one that has been valued most by all peers (i.e., $b_x = \max_i b_{i,x}$, $b_y = \max_i b_{i,y}$, $b_z = \max_i b_{i,z}$), then no payment is demanded, as there was no real conflict to be solved. Note also that, in case of intra-profile conflicts, the payment is always zero, as the winning policy is never 'imposed' on anyone, that is, there is no added benefit over anyone. The rationale for this payment scheme is that applications are not paying for the resources they use when applying a policy, but, rather, for the (added) quality-of-service level they get from it. We assume that ties are broken by selecting a policy randomly (i.e., a k-way tie is decided by flipping a 'k-sided coin', where each policy is chosen with probability $1/k$).

If a service *sname* is requested which requires the cooperation of a set of peer *peerList*, then the whole conflict resolution mechanism can be summarised as follows:

$$\mathcal{G} : command \rightarrow \mathrm{P}$$
$$\mathcal{G}[\![sname\ peerList]\!] = \mathcal{W}\ [\![\ \mathcal{B}\ [\![\ \mathcal{I}[\![sname]\!]_{\{peerList\}}\ ]\!]_{\{peerList\}}\ ]\!]$$

A service request may then produce one the following two results:

- $\mathcal{G}[\![sname\ peerList]\!] = pname$: service *sname* is delivered using policy *pname* (either because all peers agreed on the policy, or because *pname* was the policy selected during a conflict resolution process);
- $\mathcal{G}[\![sname\ peerList]\!] = \epsilon$: the service request fails as no policy can be found that is both agreed on by all peers and valid in the current context.

The auctioning mechanism has been described in the general situation where there are different applications running on different hosts (inter-profile conflict). N-on-1 conflicts are detected and solved in the same way as inter-profile conflicts. However, as the application instances involved are running on the same host (i.e., in our model, on the same middleware instance), no communication overhead is required, and the solution set P^* can be computed locally. Intra-profile conflicts can be considered a degeneration of inter-profile conflicts, where the number n of bidders is 1, and the solution set coincides with P_1 (i.e., the set of policies that can be applied in the current execution context, according to $peer_1$ application profile). The auction proceeds as described above, selecting the policy that maximises $peer_1$ utility, without communication costs.

Once a conflict has been detected and resolved using the auctioning mechanism presented above, no further action is taken. The conflict cannot be removed as it is usually not local to a profile but distributed among the profiles of different peers. If the peers involved change, or if the context changes, there may be no conflict at all. Also, we assume that each auction is carried out in isolation: we cannot assume that next time the same conflict arises, the winning policy will be the same one, as the result depends on current peer quotas, utility functions and application profiles. Therefore, each conflict resolution process stands alone.

There are few questions that need to be answered about the process described above; in particular, how is an utility function defined, and how is the quota managed by middleware? We answer these questions in the following sections.

B. Utility Function

Whenever a conflict is detected, either inside a profile (intra-profile conflict) or among various profiles (inter-profile conflict), user goals, such as privacy of information for the messaging service, must be taken into account. In other words, users should be allowed to influence the conflict resolution process operated by the middleware as they are the only ones who know what their goals are at the moment, and how different outcomes are valued.

Utility functions serve this purpose. A utility function u_i translates user goals with respect to peer $peer_i$ into a value $u_{i,j}$, that represents the price the user is currently willing to pay to have policy p_j applied, that is, to see its goals fulfilled. The following holds:

$$u_{i,j} \geq 0,\ \forall i \in [1,n],\ j \in [1,m].$$

As in 'human' auctions, values cannot be negative; a value $u_{i,j} = 0$ means that policy p_j is not relevant to peer $peer_i$, that is, applying p_j does not give any benefit to $peer_i$ (this is a plausible 'machine' representation of a 'human' refuse to bid).

Utility functions vary *dynamically* to reflect changes in the user goals; however, the value they return is computed over *static* policy specifications which estimate the *consumption* of resources that applying the policy entails, and the *benefits* it gives in terms of quality of service. If $\mathrm{R} \subset \Sigma^*$ defines the set of resource names that the middleware monitors, and $\mathrm{Q} \subset \Sigma^*$ the set of benefits achieved by applying policies in P, then a policy specification can be described as a domain set $\mathrm{PSPEC} = \wp(\{\mathrm{R} \cup \mathrm{Q}\} \times level)$, being $level ::= \text{'1'}| \ldots |\text{'LMAX'}$ an estimate of resource consumption/benefit achieved which the policy developers compute before delivering the policy.

The abstract syntax of a utility function is given in Figure 7, where $cb\_name \in (\mathrm{R} \cup \mathrm{Q})$ is a name that uniquely identifies a resource or benefit inside a policy specification, and $weight ::= \text{'1'}| \ldots |\text{'WMAX'}$ is a factor that represents the importance the user associates to a particular resource/benefit (the higher the weight, the more important the resource/benefit). Although we consider the issue

$$
\begin{array}{rcl}
ufunction & ::= & addendList \\
addendList & ::= & addend\ addendList \mid addend \\
addend & ::= & cb\_name\ weight
\end{array}
$$

Fig. 7. Utility Function Abstract Syntax.

$$
\begin{array}{rcl}
\mathcal{U} & : & ufunction \rightarrow \text{PSPEC} \rightarrow \mathbb{R}^+ \\
\mathcal{U}[\![addend\ addendList]\!]_{ps} & = & \mathcal{U}[\![addend]\!]_{ps} + \mathcal{U}[\![addendList]\!]_{ps} \\
\mathcal{U}[\![cb\_name\ weight]\!]_{ps} & = & \dfrac{int(\mathcal{S}[\![cb\_name]\!]_{ps}) * int(weight)}{LMAX * WMAX * RQMAX}
\end{array}
$$

Fig. 8. Semantics of Utility Function. $\mathcal{S} : (\text{R} \cup \text{Q}) \rightarrow \text{PSPEC} \rightarrow level$ is a function that, given a resource/benefit name *cb_name*, and a policy specification *ps*, fetches the *level* associated to *cb_name* in *ps* (if the utility function tries to retrieve a value for a resource/benefit that does not appear in the policy specification, the returned value is 0). *int* is a function that given a literal in $\{'1', \ldots, '\text{MAX}'\}$, returns the corresponding integer value in $[1, \text{MAX}]$; $LMAX{*}WMAX{*}RQMAX$ is the maximum bid an application can place, being $RQMAX$ the maximum number of resources/benefits of interest to an application.

of generating weights that represent user needs as faithfully as possible a matter of future research, we will give a flavour of how these numbers can be obtained from users and directly used in our system in Section V-B.

Whenever a peer $peer_i$ is involved in a bidding process, its utility function is retrieved and used to find the peer utility value $u_{i,j}$ for each conflicting policy p_j. The semantics of a utility function is presented in Figure 8. As shown, each value is normalised to vary in a range $[0, 1]$, so that different bids can be compared effectively, and money fairly redistributed (see Section IV-C).

As stated before, while policy specifications are fixed, utility function specifications change over time, as they have to reflect current user needs. This implies that the reflective mechanism of our middleware has to allow dynamic modification of utility function specifications. This allows our conflict resolution scheme to fulfil our second requirement, that is, *customisation*.

Note that, to avoid incompatibility among the prices bid during a conflict resolution process, utility functions are locked at the beginning of an auction, and cannot be modified until the auction finishes. Thus, applications cannot 'cheat' and associate high bids to the policies they value most, while bidding zero for the others, to increase the chances to have the policy they value most finally applied, as this would require applications to change the weights of their utility functions during the auction.

C. Quota Allocation

When describing the rules of our mechanism (see Section IV-A), we specified that each peer $peer_i$ is allowed to bid a value $b_{i,j}$ for policy p_j, given that this value is lower than its current quota q_i. We now explain how this quota is managed.

Whenever an application instance A_i is started, an initial quota $q_i = q_{init}$ is awarded. Each time A_i participates to an auction, its quota is decreased by an amount equal to $f_i \in [0, 1]$, as defined in Figure 25 in the Appendix. A_i's underlying middleware instance collects A_i payments and stores them in a wallet $\overline{q}(i)$. We assume that there is no flow of money from one middleware instance to another (i.e., each application instance pays its underlying middleware instance). Moreover, we assume that there is no explicit utility transfer among applications (e.g., no money can be transfered to a peer to compensate for a disadvantageous agreement).

Every t time units, each middleware instance redistributes the money it has collected in its wallets $\overline{q}(i)$, $i \in [1, n]$, to the various application instances A_i, $i \in [1, n]$. The amount of money each application instance gets back is in direct relation to the number of interactions it has been involved during the last t time units, and in inverse relation to the amount of money it bid. We define an interaction as a service request which incorporates an inter-profile conflict (intra-profile conflicts are excluded from the quota recharging as no flow of money occurs).

In particular, if we indicate with $N_t(i)$ the number of interactions in which application instance A_i was involved in the last t time units, then the recharging process is carried out as described below:

$$q_i = q_i + \left(\overline{q}(i) - \frac{\overline{q}(i)}{N_t(i)}\right); \qquad \overline{q}(i) = \frac{\overline{q}(i)}{N_t(i)}$$

being $\overline{q}(i)$ the money currently stored by the middleware in the wallet associated to A_i, and q_i A_i current quota.

This quota redistribution scheme discourages dictatorial interactions: if an application instance bids very high in a few interactions, 'imposing' its preferred policy over the others, then only a very low amount of money is returned during a recharging process. The only way to get money back from the middleware is to participate in other interactions in a more cooperative fashion (i.e., by bidding lower and interacting more). For example, let us assume that at time t_0, two application instances A_1 and A_2 are started and awarded the same quota $q_i = 3$, $i \in \{1, 2\}$. During the following t time units, they are involved in a number of interactions that cost them altogether the same amount of money; however, while A_1 bid aggressively, paying a lot of money in few interactions, A_2 was more cooperative, paying low amounts in many interactions. As a result, our quota redistribution scheme returns money to A_2 faster than to A_1 (see Figure 9).

The approach to quota redistribution that we have described could be defined as 'conservative': at any time, an application instance A_i has got the same amount of money, although split differently between its current quota q_i and the corresponding middleware wallet $\overline{q}(i)$. In other words:

$$\overline{q}(i) + q_i = q_{max}$$

being q_{max} a fixed amount that is the same for any application. At time t_0 when an application instance A_i is started, different choices of q_{init} and $\overline{q}(i)$ are possible. In

Time / Action	q_1	$\overline{q}(1)$	q_2	$\overline{q}(2)$
t_0 / Start	3	0	3	0
t_1 / Bid	2.1	0.9	2.7	0.3
t_2 / Bid	1.2	1.8	2.4	0.6
t_3 / Bid			2.1	0.9
t_4 / Bid			1.8	1.2
t_5 / Bid			1.5	1.5
t_6 / Bid			1.2	1.8
t_7 / Redistribution	**2.1**	0.9	**2.7**	0.3

Fig. 9. Example of Quota Redistribution (with $t_7 - t_0 = t$).

```
talkReminder
    soundAlert
        location = outdoor          battery    2
    vibraAlert                      privacy    10
        location = indoor           focusing   10
    silentAlert
        location = indoor
        battery < 15%
            (a)                          (b)
```

Fig. 11. (a) Application Profile. (b) Utility Function. $peer_1$ aims at maximising privacy and focusing, without too much interest in battery consumption.

particular, any assignment that complies with the following equations is acceptable:

$$\forall \alpha \in [0,1] \begin{cases} q_{init} = \alpha \cdot q_{max} \\ \overline{q}(i) = (1-\alpha) \cdot q_{max} \end{cases}$$

Setting $\alpha = 1$ favours newly started application instances, while setting $\alpha = 0$ favours applications that have been executing for a long while. The differences among these possibilities disappear while time passes. It is beyond the scope of this paper to investigate the optimal choice for q_{init}, q_{max}, t and α.

This concludes the discussion about our auctioning approach to the conflict resolution problem. In the following section, we illustrate how this mechanism can be instantiated and used to solve conflicts.

D. Conference Application

In this Section, we present examples of intra- and inter-profile conflicts that may occur in the conference application, and show how our auctioning mechanism is used to resolve them.

D.1 Intra-profile conflict: Talk Reminder

Let us assume that the talk reminder service can be delivered using one of the following policies: a `soundAlert` policy, a `vibraAlert` policy, and a `silentAlert` policy. Each of these policies requires different amounts of resources to be used (in particular, battery), and achieves a different quality of service (in terms of focusing and privacy). The corresponding policy specifications are shown in Figure 10.

```
soundAlert: {(battery,6),(privacy,1),(focusing,8)}
vibraAlert: {(battery,10),(privacy,7),(focusing,8)}
silentAlert: {(battery,1),(privacy,10),(focusing,2)}
```

Fig. 10. Policy Specifications.

Whenever a talk reminder has to be delivered, the application profile is consulted to find out which policy to apply. Let us assume that the application profile is the one illustrated in Figure 11(a), and that the talk reminder service is invoked when the user is attending a talk (i.e., `location = indoor`), and battery is lower than 15%, so that both `vibraAlert` and `silentAlert` are enabled (intra-profile conflict). Note that, although it could be argued that such a conflict would not exist if the profile were properly written (i.e., if a line containing `battery > 15%` were added to the context of the `vibraAlert` policy), avoiding context overlaps is not so easy. When the number of resources associated to a context increases, chances of making mistakes and of writing profiles with context overlaps increase quickly. As already argued, a static conflict analysis would be unmanageable on portable devices, and therefore a dynamic solution is needed. We now illustrate how our dynamic conflict resolution mechanism works effectively to solve this conflict, assuming that the utility function is the one illustrated in Figure 11 (b).

Computation of the solution set. First, the solution set P^* is computed; as only one peer is involved, P^* coincides with P_1:

$$\mathcal{I}[\![\texttt{talkReminder}]\!]_{\{peer_1\}} = \{\texttt{vibraAlert}, \texttt{silentAlert}\}$$

Computation of bids. High weights associated to resources in utility function specifications mean that the user aims at sparing resources; however, policy specifications estimate the amount of resources consumed, not spared. In order to give higher scores (i.e., higher bid prices) to the policies that reduce resource consumption, we therefore need to compute the value: LMAX− *expected consumption*. For example, if we assume LMAX = 10, WMAX = 10, and RQMAX = 5 (i.e., battery, bandwidth, focusing, availability and privacy), then:

$$u_{peer_1}(\texttt{vibraAlert}) = \frac{(10-10)*2 + 7*10 + 8*10}{10*10*5} = 0.3$$

Assuming that the peer quota $q_{peer_1} > 1$ (i.e., the bid is not constrained by current quota, as each bid $b_{1,j} \in [0,1]$), we obtain:

$$\mathcal{B}[\![\{\texttt{vibraAlert}, \texttt{silentAlert}\}]\!]_{\{peer_1\}} = \{(\texttt{vibraAlert}, peer_1, 0.3), (\texttt{silentAlert}, peer_1, 0.276)\}$$

Election of the winner. As only one peer is involved in an intra-profile conflict, maximising social welfare coincides with maximising individual utility. The winning policy is the one that $peer_1$ valued most and no quota adjustment is needed.

$$\mathcal{W}[\![\, \mathcal{B}[\![\{\texttt{vibraAlert}, \texttt{silentAlert}\}]\!]_{\{peer_1\}}]\!] = \texttt{vibraAlert}$$

D.2 Inter-profile conflict: Messaging

Peers can exchange messages using one of the following policies: a `charMsg` policy, a `plainMsg` policy,

```
charMsg: {(battery,4),(bandwidth,10),(availability,10)}
plainMsg: {(battery,3),(bandwidth,6),(availability,7)}
compressedMsg: {(battery,5),(bandwidth,4),(availability,5)}
encryptedMsg: {(battery,6),(bandwidth,7),(availability,4),
              (privacy,10)}
```

Fig. 12. Policy Specifications.

```
% peer 1
messagingService
    charMsg
        bandwidth > 70%
    plainMsg
        bandwidth < 70%
    compressedMsg
        bandwidth < 35%
    encryptedMsg
        battery > 50%

% peer 2
messagingService
    plainMsg
        battery < 50%
    compressedMsg
        bandwidth < 40%

% peer 3
messagingService
    plainMsg

    compressedMsg
        bandwidth < 40%
    encryptedMsg
        battery > 60%
```

Fig. 13. Application Profiles.

```
% peer 1              % peer 2          % peer 3
battery        4      battery    7     privacy  10
bandwidth      3      bandwidth  9
availability   10
```

Fig. 14. Utility Functions. $peer_1$ aims at maximising availability without wasting resources; $peer_2$ aims at minimising resource consumption, and $peer_3$ aims at maximising privacy.

a `compressedMsg` policy, and an `encryptedMsg` policy. Again, each of these policies requires different amounts of resources (in particular, battery and bandwidth), and achieves a different quality of service (in terms of availability and privacy of the message). The corresponding policy specifications are shown in Figure 12.

Let us suppose that three peers $peer_1$, $peer_2$, and $peer_3$ are in reach of each other and want to start a chat. In order to do so, they have to agree on a common policy to be applied to exchange messages. During the lifetime of the chat, the policy used may change to adapt to new context configurations where the currently used policy is no longer suitable. However, when this happens, all the chatting peers must agree on the new policy to use.

The peers' application profiles are represented in Figure 13. Note that $peer_3$ leaves the `plainMsg` policy always enabled: this is a good way to reduce the risk of ending a conflict resolution process with a failure because no agreed policy could be found. However, this increases the risk of conflicts and, consequently, the time used to resolve them (which is anyway rather low, as it will be shown in Section V-A). It is up to the application to decide which strategy is best.

Assuming that the utility functions are the ones shown in Figure 14, and that the current execution context enables the following sets of policies:

$$\begin{aligned} P_1 &= \{\texttt{plainMsg}, \texttt{compressedMsg}, \texttt{encryptedMsg}\} \\ P_2 &= \{\texttt{plainMsg}, \texttt{compressedMsg}\} \\ P_3 &= \{\texttt{plainMsg}, \texttt{compressedMsg}, \texttt{encryptedMsg}\} \end{aligned}$$

for peers $peer_1$, $peer_2$ and $peer_3$ respectively, then the conflict resolution process proceeds as described below.

Computation of the solution set. First, the solution set P^*, that is, the set of commonly agreed policies is computed:

$$\begin{aligned} \mathcal{I}[\![\texttt{messagingService}]\!]_{\{peer_1,peer_2,peer_3\}} &= P_1 \cap P_2 \cap P_3 \\ &= \{\texttt{plainMsg}, \texttt{compressedMsg}\} \end{aligned}$$

Computation of bids. Assuming, as before, LMAX = 10, WMAX = 10, RQMAX = 5, and that each peer has a quota $q_{peer_i} > 1$, we obtain:

$$\begin{aligned} &\mathcal{B}[\![\{\texttt{plainMsg}, \texttt{compressedMsg}\}]\!]_{\{peer_1,peer_2,peer_3\}} = \\ &\{(\texttt{plainMsg}, peer_1, 0.22), (\texttt{compressedMsg}, peer_1, 0.176), \\ &(\texttt{plainMsg}, peer_2, 0.17), (\texttt{compressedMsg}, peer_2, 0.178), \\ &(\texttt{plainMsg}, peer_3, 0), (\texttt{compressedMsg}, peer_3, 0)\ \} \end{aligned}$$

Election of the winner. Bids received for each policy in the solution set are added, and the policy that maximises the sum (i.e., social welfare) is selected.

	plainMsg		compressedMsg	
$peer_1$	0.22	+	0.176	+
$peer_2$	0.17	+	0.178	+
$peer_3$	0		0	
	0.39		0.354	

$$\mathcal{W}[\![\ \mathcal{B}[\![\{\texttt{plainMsg}, \texttt{compressedMsg}\}]\!]_{\{peer_1,peer_2,peer_3\}}]\!] = \texttt{plainMsg}$$

Finally, each peer quota is adjusted in the following way:

$$\begin{aligned} q_1 &= q_1 - \tfrac{0.22-0.17}{1} - \tfrac{0.17-0}{2} - \tfrac{0}{3} \\ q_2 &= q_2 - \tfrac{0.17-0}{2} - \tfrac{0}{3} \\ q_3 &= q_3 - \tfrac{0}{3} \end{aligned}$$

V. Evaluation

In this section, we evaluate our approach in terms of performance and usability.

A. Performance

The performance of CARISMA have been measured based on our current implementation: the middleware has been implemented in Java using jdk 1.4.1, while application profiles and utility functions have been encoded using the eXtensible Markup Language [12] (their grammar has been defined in two associated XML Schema available at http://www.cs.ucl.ac.uk/staff/l.capra/schema). We chose to use XML as we believe this language may enhance context-aware and user-driven interactions between middleware and applications, supporting a representation of

information that can be both easily manipulated by machines, and readily understood by humans. Also, XML related technologies, in particular, DOM [13] and XPath [14], and available XML parsers [15] have considerably reduced the development time. Communication takes place via a simple message passing mechanism that we have implemented. The middleware platform currently requires only 110Kb of persistent storage, and less than 800Kb of memory (without considering the memory required by the Java Virtual Machine and XML parser).

We performed tests on Dell Latitude laptops equipped with 128MB RAM, Intel Pentium II processors rated at 300MHz, and connected in an ad-hoc network using Cisco Aironet 340 10Mbps wireless cards. The operating system used was Microsoft's Windows2000 and the Java Virtual Machine version was 1.4.1. We believe these machines are well-suited to estimate the performance of our middleware, as they do not outperform the currently available portable devices (e.g., the Sony Ericsson P800 mobile phone is equipped with 12Mb internal storage, plus external memory stick, and ARM9 200MHz processor; the HP iPAQ Pocket PC h5450 is equipped with 64Mb RAM and Intel 400MHz processor; COMPAQ Tablet PC TC1000 is already extremely powerful, with 256MB RAM minimum and 1GHz processor).

In order to estimate the scalability of CARISMA in terms of the number of devices involved in the delivery of a service, the number of (possibly conflicting) policies associated to each service, the number of contexts for each policy, and the number of resources in each context, we have implemented a benchmark that allowed us to tune each of these parameters independently. The charts shown in this section represent the average of the results obtained over 20 service requests.

In the remainder of this section, we first consider a simple local service, i.e., a service that involves a single device. In this simple scenario, we illustrate the basic overheads introduced by reflection, by context-awareness while varying number of contexts and resources associated to each policy, and by the conflict resolution mechanism. We then move to a distributed setting and we analyse the performance of CARISMA while varying the number of devices involved in a service request.

A.1 Impact of Reflection

Figure 15 illustrates the overhead introduced by reflection over a basic mechanism where a service is statically associated to a policy. This lower bound is represented in the picture by the intersection of the curve with the Y axis. As the picture shows, the overhead (in milliseconds) is more or less linear in the number of policies associated to the service, and is kept below 1 second even when 10 policies are associated to the same service. This overhead includes also the evaluation of a simple context configuration made of one context with one resource associated to each policy (these associations are necessary to avoid conflicts).

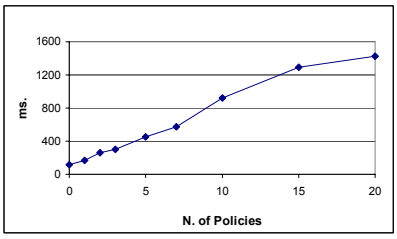

Fig. 15. Impact of Reflection.

A.2 Impact of Context-Awareness

Fixing the maximum number of policies associated to a service to 10, Figure 16 shows the impact of context-awareness on performance. We assume here that, whenever a service is invoked, the current value of each resource is already available (i.e., the middleware is probing the physical sensors at regular intervals to keep updated context information). The performance results we are discussing do not consider the time necessary to initialise a sensor and to process the information gathered through it; we believe this approach is plausible, as sensors may greatly vary in nature, and therefore may introduce overheads of different orders of magnitude (e.g., knowing the amount of battery left requires much less time than gathering and processing location information).

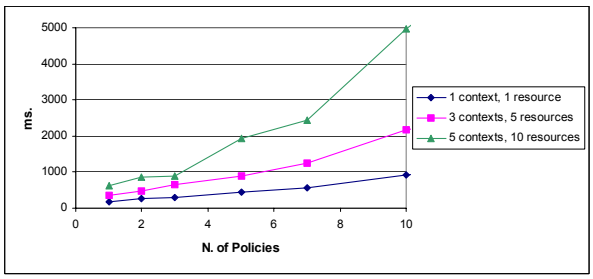

Fig. 16. Impact of Context-awareness.

As shown, having five or more contexts for each policy, and ten or more resources for each context, represents a scalability limit in the performance of CARISMA. This is due to the fact that the number of comparisons between the current context and the associations encoded in the profile grows exponentially with the number of contexts and resources. In our experience with the conference application, however, this scalability limit was never reached, as having five policies associated to three contexts with five resources each, already represented the maximum level of adaptation we needed (i.e., the worst-case scenario); in this case, the average amount of time to request a local service is still below one second.

A.3 Impact of Conflicts

As the following two figures show, the conflict resolution process has a minor impact on the performance of CARISMA. First (see Figure 17), the number of utility function parameters does not influence the performance of

a service request at all. Also, this chart depicts the performance of a local service request where no context is associated to the policies (i.e., they are always enabled); comparing this chart with the one shown in Figure 15, we can conclude that the conflict resolution mechanism introduces a much lower overhead than the simplest case of context-awareness. In fact, it takes about 900ms. to determine which policy to apply out of ten, in case a very simple, mutually exclusive (i.e., no conflict) context is provided (one context with one resource), while it takes less than 400ms. in case no context is provided and the conflict resolution procedure has to be executed.

Second, the conflict resolution mechanism adds a negligible overhead over the standard mechanism (inclusive of context-awareness), as depicted in Figure 18. In case each policy is associated with the same number of contexts and resources, the overhead introduced by the conflicts resolution mechanism is almost constant and in the order of 200ms.

We can conclude that a good strategy in developing applications on top of CARISMA is to associate only minimal context configurations to the policies, and have the auction mechanism solve potential conflicts.

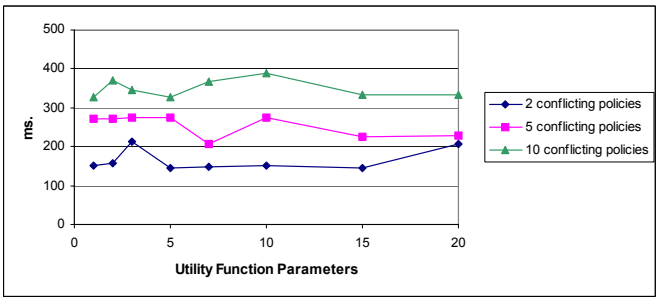

Fig. 17. Impact of Utility Function Parameters.

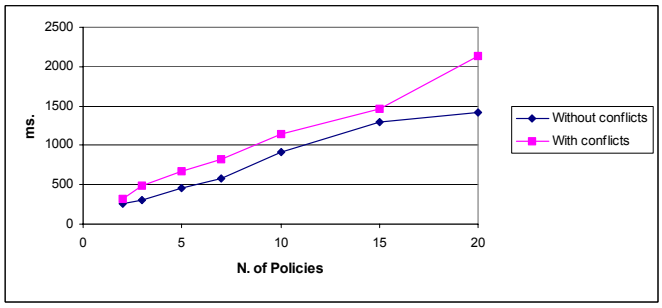

Fig. 18. Impact of Conflict Resolution Mechanism.

A.4 Impact of Distribution

The last chart shows the performance of CARISMA in answering a service request for two plausible profile configurations, while varying the number of devices involved in the delivery of the service. These results have been computed considering an implementation of the auction protocol that is based on the 3-step communication protocol shown in Figure 19.

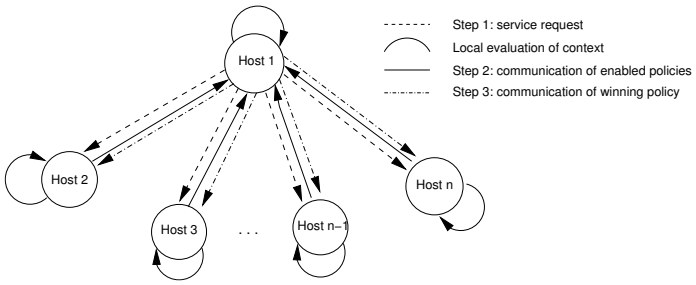

Fig. 19. Communication Protocol.

This protocol tries to maximise performance by parallelising the context evaluation step among the n peers involved in a service execution. Whenever an application running on Host 1 needs a service that requires the cooperation of $n-1$ other applications running on as many hosts, these steps are followed.

Step 1: first, Host 1 sends out a service request message to each of the $n-1$ peers involved in the service execution. At this point, all the n peers evaluate their local context in parallel, and find out the locally enabled policies (i.e., the sets P_i, for $i \in [1, n]$). Moreover, although no conflict has been detected yet, they compute a bid for each of these policies.

Step 2: the $n-1$ peers communicate their own P_i and corresponding bids back to the requesting host, which now computes the solution set P^*. If no conflict is found, the pre-computation of the bids was a waste of time and resources, but if, on the contrary, a conflict is detected, two additional communication steps are saved (i.e., to ask the $n-1$ peers to bid for the policies in P^*, and to communicate these bids back to the requesting host). As the time taken by the computation of the bids is negligible (i.e., few milliseconds), compared to the time taken by two additional communication steps, the pre-computation proves to be worthwhile.

Step 3: once the winning policy has been selected, and the payments have been computed, the requesting host sends this information to the $n-1$ participating peers, and the service can be finally delivered.

Individual failures of participating peers, taking place during the protocol execution, do not compromise its success, as long as there are at least $0 < m < n$ peers connected until the end of the process (the minimum number of connected peers, m, is application dependent). However, if the requesting peer fails, the entire service request is aborted.

As shown in Figure 20, the overhead tends to be constant, and does not increase considerably while increasing the number of devices involved. The results shown here do not consider peer failures; in case failures are taken into account, the overhead depends on the timeout values used before acknowledging a peer is no more in reach.

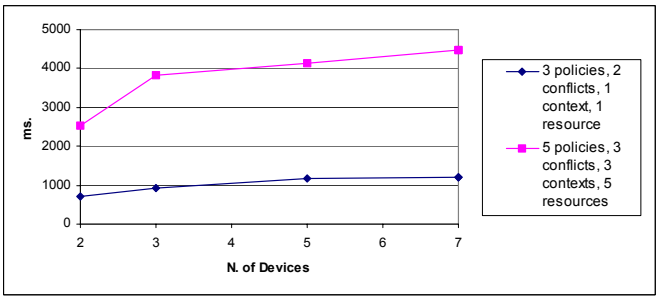

Fig. 20. Impact of Conflicts in a Distributed Setting.

B. Usability

To estimate the usability of our middleware, we have assigned a student, with little computer science background, the task of implementing the conference application on top of CARISMA. From the student's report, it emerged that the most difficult task was to decide which non-functional parameters the user could tune, and how to map them into application profiles, while using the abstractions and mechanisms provided by CARISMA turned out to be rather straightforward.

The student decided to allow the end-user of the system to tune the importance he/she assigned to both non-functional requirements (i.e., availability of information, accuracy and privacy), and to local resources (i.e., memory, battery and bandwidth), by means of the customisation windows illustrated in Figure 21. The effort required from the end-user of the system was rather limited: when his/her preferences were changing, all he/she had to do was to input the new preferences through these windows.

Based on these preferences, the student implemented a synthesising algorithm to write both application profiles and utility functions. Application profiles were encoding associations with 2/3 policies per service, each with 1/2 contexts made of 2/3 resources. Utility functions simply listed the importance users assigned to customisable parameters (e.g., in the picture above, `memory=4`, `battery=9`, `bandwidth=0`, `availability=8`, `accuracy=3`, `privacy=0`).

Estimating the end-user efforts in teaching the system to behave according to his/her own preferences, strongly depends on the level of adaptation the application permits: the higher the number of parameters that are subject to customisation, the finer the level of adaptation the system

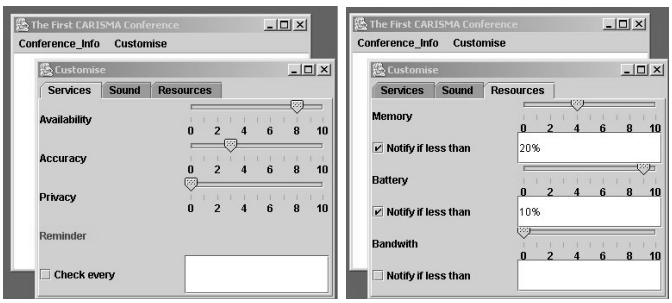

Fig. 21. Conference Application Customisation.

may achieve. However, this would have an impact on the amount of human effort required, as well as on the complexity of the synthesising algorithm that application engineers have to come up with. Note that these issues are not intrinsic in our middleware model, but apply, in general, to scenarios where adaptation to changing context and user requirements is needed; in developing context-aware applications, therefore, granularity of adaptation has to be traded against human effort. Further research is needed in this direction to estimate where the equilibrium lies.

VI. Related Work

Providing a detailed review of the state of the art in the area of context-awareness, reflection, and mobile computing is beyond the scope of this paper. A critical literature review in the area of reflection and context-awareness (and, more generally, in middleware for mobile computing), can be found in [16]. In this paper, we compare our work with what various research communities have done, as far as conflict resolution is concerned.

The operating systems community has studied the issue of conflicts in a distributed environment, where conflicts manifest themselves as processes competing for shared resources. Microeconomic techniques, and auctions in particular, have been explored; in [17], a market-like bidding mechanism is described which assigns tasks to processors that have given the lowest estimated completion time; similar techniques have been used to manage network traffic [18] and allocation of storage space [19]. We have demonstrated that game theory can also be successfully used to resolve QoS conflicts that arise in the mobile setting; however, the nature of conflicts is fundamentally different, thus requiring different conflict resolution algorithms. In particular, resource conflicts happening at the operating system level represent competitive situations where only one competitor obtains the resources, leaving all the others without them. In our case, instead, collaboration characterises the nature of the auction better: peers participating in the delivery of a service will *all* get the good (the delivery of the service), but with varying degrees of satisfaction. Traditional auctions cannot be applied in this setting, and we had to come up with a novel mechanism to deal with these conflicts.

Despite the extensive research that has been carried out within the mobile middleware community, the issue of QoS conflicts has attracted little attention. On one hand, many systems do not support dynamic adaptation of middleware behaviour, and thus they avoid the problem of conflicts a priori. On the other hand, systems which exploit reflection to improve flexibility and allow dynamic reconfigurability of the middleware [20][21] generally target a stationary distributed environment, where context changes (and, consequently, adaptation of middleware behaviour) are much less frequent than in a mobile setting, so that the problem of conflicts is less pressing. Data conflicts have been investigated more extensively instead: in order to maximise data availability in mobile settings, where sudden disconnections may happen frequently, even for long periods of time, sys-

tems such as Coda [22], Bayou [23] and Xmiddle [24] give users access to replicas. They differ in the way they ensure that replicas move towards eventual consistency, that is, in the mechanisms they provide to detect and remove conflicts that naturally arise in mobile systems. Data conflicts, however, are fundamentally different from the QoS conflicts we treat, and therefore these solutions can hardly be applied; in particular, inter-profile conflicts are not intrinsic in any profile, but manifest themselves only in relation to (some) other profiles, and in particular contexts, and therefore cannot be removed, but only dynamically solved.

The software engineering community has investigated the issue of conflicts too. Software development environments [25][26] have devised mechanisms for specifying consistency constraints between artifacts. They are able to detect static violations of these constraints and resolve them automatically (e.g., by propagating changes to dependent documents). Inconsistencies are often found in requirements documents, indicating conflicts between the different stakeholders involved. Requirements management methods and tools therefore include inconsistency detection and resolution mechanisms. The KAOS method [27] uses a goal-oriented approach to decompose requirements and formalises them using a temporal logic. Conflicts are detected by reasoning about the temporal logic formulae and conflict resolution strategies [28] can be applied so that requirement conflicts are not come down to design. Other requirements engineering approaches [29] leave inconsistencies in specifications and use an appropriate logic to continue reasoning, even in the presence of an inconsistency. These approaches, however, are of limited use in a mobile setting where the nature of conflicts is such that they cannot be detected statically at the time an application is designed but, instead, they can only be detected and resolved at run-time. Also, they must be resolved, otherwise applications cannot execute.

Our work is more closely related to approaches that monitor requirements and assumptions during the execution of systems. Fickas and Feather's approach towards requirements monitoring [30] uses a Formal Language for Expressing Assumptions (FLEA). FLEA is supported by a CLISP-based run-time environment, which can alert requirement violations to the user. For mobile systems, however, this is insufficient and a more proactive approach to resolving conflicts is required. Robinson and Pawlowski [31] have developed a so-called "requirements dialog meta-model", which supports not only the definition and monitoring of goals, but also the re-establishment of a dialog goal in case of a goal failure. Goal monitoring is performed actively, so that violations are detected immediately; however, this requires a consumption of resources that hand-held devices cannot bear.

In the Distributed Artificial Intelligence (DAI) community, game theory [7] has been extensively applied to treat negotiation issues. Negotiation mechanisms have been used both to assign tasks to agents, to allocate resources, and to decide which problem solving tasks to undertake (e.g., [32] [33]). These scenarios typically involve a group of agents operating in a shared environment. Each agent has its own private goal; a negotiation process is put in place that, through a sequence of offers and counter-offers, explores the chance for agents of achieving their (possibly conflicting) goals, at the lowest cost. Despite similarities with our scenario, there are a number of assumptions that differentiate our work from previous results obtained in the DAI community. In particular, in DAI the quality of the result is valued much more than the cost of achieving it; as a consequence, negotiation mechanisms are usually iterative processes which carry on until an (optimal) agreement is reached. In a mobile setting, instead, resource constraints call for simple conflict resolution mechanisms that do not waste (scarce) resources. Moreover, the nature of goals is fundamentally different. In DAI, a goal can be seen as a task composed of atomic operations that the negotiation mechanism is able to assign to different agents; in our setting, goals are rather indivisible units that suggest the quality of service levels that applications are wishing to achieve to the middleware.

Also relevant to our work is the research on quality of service provision in a mobile computing environment [34]. QoS requirements are defined by all applications and a negotiation mechanism is put in place to reach an agreement between all parties; as a result of context changes, a dynamic renegotiation of the contract may be necessary. The approaches we have analysed usually target a specific domain (e.g., multimedia applications over broadband cellular networks), mainly focusing on bandwidth allocation [35]. Moreover, applications have a rather limited way of influencing the policies that are chosen to meet QoS requirements. Our middleware aims at being general and uses reflection to give applications the power to influence the way adaptation is achieved. This may lead to disagreements among applications to reach the quality-of-service level they wish.

VII. Conclusion and Future Work

The increasing popularity of portable devices and recent advances in wireless networking technologies are facilitating the engineering of new classes of applications, which present challenging problems to designers. To accommodate the new requirements of mobility, and, in particular, the need for context-awareness and adaptation, middleware platforms for mobile computing must be capable of both deployment-time configurability and run-time reconfigurability.

In this paper we have described CARISMA, a mobile computing middleware that exploits reflective techniques to enable mobile application designers to address these requirements. Besides enabling dynamic adaptation to context, reflection may also cause conflicts. We have demonstrated how CARISMA uses micro-economic techniques effectively in order to solve conflicts that arise in the mobile setting. In particular, we have modelled a mobile distributed system as an economy, where applications compete to have a common service delivered according to their preferred quality-of-service level; in this economy, the mid-

dleware plays the role of an auctioneer, collecting bids from applications and selecting the policy that maximises social welfare. This approach is particularly suited in the mobile setting as it meets the requirements of dynamicity, simplicity and customisability that are typical of this environment. We have evaluated CARISMA in terms of performance and usability, and the results obtained have confirmed the suitability of our middleware in the domain we target.

Future improvements and extensions of CARISMA span towards different directions. Despite being a very powerful means, reflection enables adaptability and flexibility only in those contexts that middleware designers have considered likely to be unstable at design time. However, in a mobile ad-hoc setting, mobile hosts cannot forecast all the possible contexts they are going to encounter, and therefore which protocols (i.e., behaviours) they are going to need; new behaviours may be delivered from time to time to cope with unforeseen context configurations and new application needs. A future direction of research is to exploit mobile code techniques to overcome this limitation, by downloading new protocols either from a service provider or from other peers in reach which use the same behaviour [36] [37]. Moreover, only a minimum set of behaviours can be stored on a device, so to avoid wasting memory; by exchanging information about what services, code and resources are available with other peers, different behaviours can be downloaded only when needed (*if* needed). Reflection can be combined with mobile code techniques to allow applications to select from where to download protocols based on application-specific information (e.g., trusted hosts, quality of service, etc.).

To accommodate dynamicity requirements, services and policies are installed and uninstalled on the fly; moreover, different application needs result in different system configurations, that vary over time. The changing interactions among distributed services and policies may alter the semantics of the applications built on top of our reflective middleware. The development of safe customisable middleware becomes, therefore, an issue. A first step towards the definition of a formal semantics for specifying and reasoning about the properties of, and interactions among, middleware components can be found in [38]. These principles have been used, for example, in [39] to manage changes in large-scale distributed systems while ensuring application QoS requirements. The principles they use are based on a two-level architecture where the application, at the base level, interacts with the middleware, at the meta-level, via middleware-defined core services that are then used to initiate other activities. The similarity of this approach with our architecture makes us think that similar principles could be investigated to develop a formal semantics of composition within our reflective middleware framework.

Currently, each host has a local view of context; an interesting extension would be to enable each host to have a broader view of its environment and allow it to gather context information from any peer directly (or indirectly) connected to it. In [40] [41], a middleware that tackles this issue has been presented. The new definition of context presented can be easily integrated in CARISMA; there are, however, open questions that still need answering, related to the binding and re-binding of external sensors.

Another direction of research that is worth mentioning is service discovery. Traditional naming and trading service discovery techniques developed for fixed distributed systems cannot be successfully applied in mobile settings, where intermittent rather than continuous network connection is the norm. However, service discovery for mobile settings has not yet gained significant attention. Two notable exceptions are the Jini specification [42] and the work by Handorean and Roman [43]. A disadvantage common to both approaches is that they do not take quality of service requirements into account when deciding which service to use. We believe that QoS-aware service discovery would fit naturally in our framework, where application needs are made explicit and used to decide how a service should be delivered in current context. Currently, these needs are taken into account only locally; a future direction of research would be to make use of this information to discover services available in an entire ad-hoc network that would deliver the user the best QoS, according to current user-specific requirements.

Last but not least, a study that puts together middleware practitioners, HCI experts and requirement elicitation experts is necessary to estimate the amount of work required from application engineers to develop context-aware applications, and from end-users to learn how to use these systems.

Acknowledgements

The authors would like to thank Zuhlke Engineering Ltd. for supporting Licia Capra; Luca Zanolin for the useful discussions we had while developing the ideas described in the paper; Ken Binmore and Pedro Rey-Biel for their insights into the microeconomic aspects of the paper; finally, we thank the anonymous reviewers of a previous version of the paper, who have produced detailed reviews and much helped to produce this new extended version.

References

[1] Sun Microsystem, Inc., "CLDC and the K Virtual Machine (KVM)," http://java.sun.com/products/cldc/, 2000.

[2] B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 85–90.

[3] ISO 10746-1, "Open Distributed Processing – Reference model," Tech. Rep., International Standardization Organization, 1998.

[4] W. Emmerich, *Engineering Distributed Objects*, John Wiley & Sons, Apr. 2000.

[5] B.C. Smith, "Reflection and Semantics in a Procedural Programming Language," Phd thesis, MIT, Jan. 1982.

[6] L. Capra, W. Emmerich, and C. Mascolo, "Reflective Middleware Solutions for Context-Aware Applications," in *Proc. of REFLECTION 2001. The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, Sept. 2001, vol. 2192 of *LNCS*, pp. 126–133.

[7] K. Binmore, *Fun and Games: a text on game theory*, Lexington: D.C. Heath, 1992.

[8] A. Mas-Colell, M. D. Whinston, and J. R. Green, *Microeconomic Theory*, Oxford University Press, 1995.

[9] L. Capra, W. Emmerich, and C. Mascolo, "A Micro-Economic Approach to Conflict Resolution in Mobile Computing," in *Pro-*

ceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, USA, Nov. 2002, pp. 31–40, ACM Press.

[10] William Vickrey, "Counterspeculation, auctions and competitive sealed tenders," *Journal of Finance*, vol. 16, no. 1, pp. 8–37, 1961.

[11] Paul Milgrom, "Auctions and Bidding: A Primer," *Journal of Economic Perspectives*, vol. 3, no. 3, pp. 3–22, 1989.

[12] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language," Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, Mar. 1998.

[13] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, "Document Object Model (DOM) Level 1 Specification," W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium, Oct. 1998.

[14] J. Clark and S. DeRose, "XML Path Language (XPath)," Tech. Rep. http://www.w3.org/TR/xpath, World Wide Web Consortium, Nov. 1999.

[15] The Apache XML Project, "Xerces Java Parser," http://xml.apache.org/xerces-j/index.html, 2000.

[16] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich, "Middleware for Mobile Computing (A Survey) ," in *Networking 2002 Tutorial Papers*. 2002, vol. 2497 of *LNCS*, pp. 20–58, Springer.

[17] T. W. Malone, Richard E. Fikes, K. R. Grant, and M. T. Howard, "Enterprise: A market-like task scheduler for distributed computing environments," in *The Ecology of Computation*, Bernardo A. Huberman, Ed., pp. 177–205. North-Holland, Amsterdam, 1988.

[18] J. Sairamesh, D. Ferguson, and Y. Yemini, "An Approach to Pricing, Optimal Allocation and Quality of Service Provisioning in High-speed Packet Networks," in *Proc. of Conference on Computer Communications*, Boston, Massachusetts, Apr. 1995.

[19] D. Ferguson, C. Nikolaou, and Y. Yemini, "An Economy for Managing Replicated Data in Autonomous Decentralised Systems," in *Proc. of International Symposium on Autonomous and Decentralised Systems*, Los Alamitos, CA, 1993, pp. 367–375, IEEE Computer Society Press.

[20] T. Ledoux, "OpenCorba: a Reflective Open Broker," in *Reflection'99*, Saint-Malo, France, 1999, vol. 1616 of *LNCS*, pp. 197–214, Springer.

[21] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An Architecture for Next Generation Middleware," in *Proc. of Middleware '98*. Sept. 1998, LNCS, pp. 191–206, Springer Verlag.

[22] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, Apr. 1990.

[23] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Cooper Mountain, Colorado, Aug. 1995, pp. 172–183.

[24] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, "XMIDDLE: A Data-Sharing Middleware for Mobile Computing," *Int. Journal on Personal and Wireless Communications*, vol. 21, no. 1, pp. 77–103, April 2002.

[25] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr, "Building Integrated Software Development Environments — Part 1: Tool Specification," *ACM Trans. on Software Engineering and Methodology*, vol. 1, no. 2, pp. 135–167, 1992.

[26] W. Emmerich, "Tool Specification with GTSL," in *Proc. of the 8^{th} Int. Workshop on Software Specification and Design, Schloss Velen, Germany*. 1996, pp. 26–35, IEEE Computer Society Press.

[27] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, pp. 3–50, 1993.

[28] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 908–926, Nov. 1998.

[29] Anthony Hunter and Bashar Nuseibeh, "Managing Inconsistent Specifications: Reasoning, Analysis, and Action," *ACM Trans. on Software Engineering and Methodology*, vol. 7, no. 4, pp. 335–367, Oct. 1998.

[30] S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," in *Proc. of the 2^{nd} IEEE Int. Symposium on Requirements Engineering, York*. 1995, pp. 140–147, IEEE Computer Society Press.

[31] William N. Robinson and Suzanne D. Pawlowski, "Managing requirements inconsistency with development goal monitors," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 816–835, 1999.

[32] G. Zlotkin and J. S. Rosenschein, "Mechanisms for Automated Negotiation in State Oriented Domains," *Journal of Artificial Intelligence Research*, vol. 5, pp. 163–238, Oct. 1996.

[33] G. Zlotkin and J. S. Rosenschein, "A Domain Theory for Task Oriented negotiation," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, AUG 1993, pp. 416–422.

[34] Dan Chalmers and Morris Sloman, "A Survey of Quality of Service in Mobile Computing Environments," *IEEE Communications Surveys*, vol. 2, no. 2, 1999.

[35] A. Campbell, "Mobiware: Qos-aware middleware for mobile multimedia communications," in *7th IFIP International Conference on High Performance Networking*, White Plains, NY, Apr. 1997.

[36] Licia Capra, Cecilia Mascolo, Stefanos Zachariadis, and Wolfgang Emmerich, "Towards a Mobile Computing Middleware: a Synergy of Reflection and Mobile Code Techniques," in *In Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, Bologna, Italy, Oct. 2001.

[37] S. Zachariadis, C. Mascolo, and W. Emmerich, "Exploiting Logical Mobility in Mobile Computing Middleware," in *Proceedings of IEEE Workshop on Mobile Team Work. Co-located with ICDCS02.* , July 2002.

[38] N. Venkatasubramanian and C. Talcott, "Meta-architectures for resource management in open distributed systems," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, Aug. 1995, pp. 144–153, ACM Press.

[39] N. Venkatasubramanian, M. Deshpande, S. Mahopatra, S. Gutierrez-Nolasco, and J. Wickramasuriya, "Design and implementation of a Composable Reflective Middleware Framework," in *Proceedings of the IEEE International Conference on Distributed Computer Systems*, Mesa, AZ, Apr. 2001, pp. 644–653, IEEE Computer Society Press.

[40] G.-C. Roman, C. Julien, and Q. Huang, "Network Abstractions for Context Aware Mobile Computing," in *Proceedings of the 24^{th} International Conference on Software Engineering (ICSE 2002)*, Orlando, Florida, May 2002, pp. 363–373.

[41] C. Julien and G.-C. Roman, "Egocentric Context-Aware Programming in Ad Hoc Mobile Environments," in *Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, Nov. 2002, pp. 21–30.

[42] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath, *The Jini[tm] Specification*, Addison-Wesley, 1999.

[43] R. Handorean and G.-C. Roman, "Service Provision in Ad Hoc Networks," in *Coordination 2002*. 2002, Springer.

Appendix

I. Semantics

$$
\begin{aligned}
\mathcal{I} &: S \rightarrow \wp(\text{PEER}) \rightarrow \wp(\text{P}) \\
\mathcal{I}[\![sn]\!]_{\{peer\ peerList\}} &= \mathcal{I}[\![sn]\!]_{\{peer\}} \cap \mathcal{I}[\![sn]\!]_{\{peerList\}} \\
\mathcal{I}[\![sn]\!]_{\{peer\}} &= \mathcal{F}[\![serv(sn, peer)]\!]_{\mathcal{E}nv(peer)}
\end{aligned}
$$

Fig. 22. Computation of the Solution Set. Given a service name sn, the semantic function $\mathcal{I}$ computes the set of policies that all peers involved in the service delivery agree.

$$
\begin{array}{rcl}
\mathcal{F} & : & service \rightarrow \mathrm{E} \rightarrow \wp(\mathrm{P}) \\
\mathcal{F}[\![sn\ policyList]\!]_e & = & \mathcal{F}[\![policyList]\!]_e \\
\mathcal{F}[\![policy\ policyList]\!]_e & = & \mathcal{F}[\![policy]\!]_e \ \cup\ \mathcal{F}[\![policyList]\!]_e \\
\mathcal{F}[\![pn\ contextList]\!]_e & = & \{pn\} \ \texttt{if}\ valid[\![contextList]\!]_e = \top \\
 & & \quad \emptyset \quad \texttt{if}\ valid[\![contextList]\!]_e = \bot
\end{array}
$$

$$
\begin{array}{rcl}
valid & : & contextList \rightarrow \mathrm{E} \rightarrow bool \\
valid[\![context\ contextList]\!]_e & = & valid[\![context]\!]_e \ \vee\ valid[\![contextList]\!]_e \\
valid[\![context]\!]_e & = & valid[\![resourceList]\!]_e \\
valid[\![resource\ resourceList]\!]_e & = & valid[\![resource]\!]_e \ \wedge\ valid[\![resourceList]\!]_e \\
valid[\![rn\ on\ valueList]\!]_e & = & eval((rn, on, valueList), e) \\
valid[\![\varepsilon]\!]_e & = & \top
\end{array}
$$

Fig. 23. Application Profile. Given a service specification *service*, the semantic function $\mathcal{F}$ evaluates, in current context $e \in \mathrm{E}$, the set of locally enabled policies. $\mathrm{E} = \wp(\mathrm{R} \times \mathrm{V})$ represents the set of all possible execution contexts (e.g, $\{(Memory, 8), (Battery, 4)\}$); *eval* is a boolean function that returns *true* if the value of resource rn in the execution context e is among the values obtained by applying the operator *on* to *valueList* (e.g., $eval((Memory, inBetween, [2, 7]), \{(Memory, 6)\}) = \top$, while $eval((Memory, lessThan, [5]), \{(Memory, 6)\}) = \bot$).

$$
\begin{array}{rcl}
\mathcal{B} & : & \wp(\mathrm{P}) \rightarrow \wp(\mathrm{PEER}) \rightarrow \wp(\mathrm{P} \times \mathrm{PEER} \times \mathbb{R}^+) \\
\mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peer\ peerList\}} & = & \mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peer\}} \cup\ \mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peerList\}} \\
\mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peer\}} & = & \bigcup_{j=1}^{m} \{(p_j,\ peer,\ \min\{q_{peer}, u_{peer,j}\})\} \\
\mathcal{B}[\![\{p\}]\!]_{\{peerList\}} & = & \{(p,\ \_\ , 0)\} \quad \text{No conflict} \\
\mathcal{B}[\![\emptyset]\!]_{\{peerList\}} & = & \emptyset \qquad\qquad\quad \text{No agreement}
\end{array}
$$

Fig. 24. Computation of Bids. Given the set of agreed policies, and the list of participating peers, the semantic function $\mathcal{B}$ associates a bid to each couple $(policy, peer)$.

$$
\begin{array}{rcl}
\mathcal{W} & : & \wp(\mathrm{P} \times \mathrm{PEER} \times \mathbb{R}^+) \ \rightarrow \mathrm{P} \\
\mathcal{W}[\![\{(p_j, peer_i, b_{i,j}),\ \forall i \in [1,n], j \in [1,m]\}]\!] & = & p_{\tilde{\jmath}} \mid \\
 & & p_{\tilde{\jmath}} \in \{\pi_1(p_j, peer_i, b_{i,j}),\ \forall i \in [1,n], j \in [1,m]\} \\
 & & \wedge \sum_{i=1}^{n} \pi_3(p_{\tilde{\jmath}}, peer_i, b_{i,\tilde{\jmath}}) \ = \max_{j \in [1,m]} \sum_{i=1}^{n} \pi_3(p_j, peer_i, b_{i,j}) \\
 & & \wedge\ pay(q_{mw}(i), f_i, q_i),\ \forall i \in [1,n] \\
\mathcal{W}[\![\{(p,\ \_\ , 0)\}]\!] & = & p \ \text{ No conflict} \\
\mathcal{W}[\![\emptyset]\!] & = & \epsilon \ \text{ No agreement}
\end{array}
$$

$$
f_i \ = \ \begin{cases}
\text{a.}\ \ 0 \text{ if } \forall k \in [1,n]\ \pi_3(p_{\tilde{\jmath}}, peer_k, b_{k,\tilde{\jmath}}) = \max_{j \in [1,m]} \pi_3(p_j, peer_k, b_{k,j}) \\
\text{b.}\ \ \sum_{\substack{l \in \{s \mid s \in [1,n] \\ \wedge\, b_{s,\tilde{\jmath}} \leq b_{i,\tilde{\jmath}}\}}} \frac{b_{l,\tilde{\jmath}} - \max(\ \{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} < b_{l,\tilde{\jmath}},\ s \in [1,n]\} \cup \{b_{min,\tilde{\jmath}}\}\ )}{\#\{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} \geq b_{l,\tilde{\jmath}},\ s \in [1,n]\} * \#\{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} = b_{l,\tilde{\jmath}},\ s \in [1,n]\}}, \\
b_{min,\tilde{\jmath}} = \min\{b_{i,\tilde{\jmath}}, i \in [1,n]\} \text{ otherwise}
\end{cases}
$$

Fig. 25. Election of the Winning Policy. Given the set of tuples $(policy, peer, bid)$, the semantic function $\mathcal{W}$ selects the policy that maximises social welfare. $\pi_i(a_1, a_2, \ldots, a_n) = a_i$ projects a tuple onto the i^{th} value; $\#\{a_1, a_2, \ldots, a_n\} = n$ computes the cardinality of a set; $q_{mw}(i)$ retrieves the quota of the middleware on top of which peer $peer_i$ is executing; finally, $pay(q_1, x, q_2) = (q_1 + x, q_2 - x)$ both increases the middleware quota q_1, and decreases the peer quota q_2, of the specified amount x.