A Micro-Economic Approach to Conflict Resolution in Mobile Computing

Licia Capra, Wolfgang Emmerich and Cecilia Mascolo
Dept. of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{L.Capra|W.Emmerich|C.Mascolo}@cs.ucl.ac.uk

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 mobile applications. These have to adapt to changes in context, such as variations in network bandwidth, exhaustion of battery power or reachability of services on other devices. We show how the construction of adaptive and context-aware mobile applications can be supported using a reflective middleware. The middleware provides software engineers with primitives to describe how context changes are handled using policies. These policies may conflict. In this paper, we classify the different types of conflicts that may arise in mobile computing. We 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 these policy conflicts can be treated. This method uses a micro-economic approach that relies on a particular type of sealed-bid auction.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, syntax*; C.2.4 [**Computer - Communication Networks**]: Distributed Systems—*Distributed applications, network operating systems*

General Terms

Design, economics, languages

Keywords

Conflict resolution, game theory, context-awareness, mobile computing, middleware, reflection

1. 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 distributed mobile applications.

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 driven and users will be reluctant to carry heavy-weight devices. Wide-area networking capabilities will continue to be based on communication with base stations, with fluctuations in bandwidth depending on physical location. In order to provide acceptable quality of service to their users, applications have to be *context-aware* and be able to adapt to context changes, such as variations in network bandwidth, exhaustion of battery power or reachability of services on other devices. Context awareness implies new requirements for the middleware that is deployed in such a mobile setting.

The aim of middleware in general is to resolve heterogeneity and distribution in order to simplify the construction of distributed systems [8]. These problems also need to be overcome in mobile computing. There are many different computing devices and application designers aim to build applications that are portable and interoperable across device types. The resolution of distribution and the provision of high-level interaction primitives are also important in a mobile setting [20]. While middleware for fixed distributed systems is largely based on the notion of transparency (i.e., distribution is hidden from both users and software engineers), it is less appropriate in a dynamic and constrained setting, such as mobile computing. It is largely agreed that different forms of middleware are needed for this scenario [17].

In [4], we have presented a mobile middleware model that supports context-aware interactions among distributed system components, based on the principles of reflection and meta-data. The middleware provides application designers with 'application profiles' that describe how the middleware realises interactions in certain contexts. Because of the highly dynamic nature of context in mobile setting, unforeseen context configurations may be entered; moreover, user's goals may vary and require different behaviours at different times. By changing, through reflection, the meta-information contained in the profiles, application designers can dynamically adapt the middleware behaviour, so as to deliver different quality of service in different context, and according to different application needs. While doing so, however, designers can create profiles that contain ambiguities, contradictions and other logical inconsistencies. We refer to these inconsistencies as *conflicts*.

The novel contribution of this paper is the design and formalisation of a microeconomic approach for *conflict resolution* that re-

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

lies on a particular type of sealed-bid auctions. In particular, our game theory-based approach treats applications as strategic players that have been programmed by different entities to pursue different goals. In order to achieve coordination under these circumstances, a mutually accepted auction protocol is established that allows applications to come to an agreement. According to this protocol, the mobile computing middleware plays the role of an auctioneer, collecting bids from applications and carrying on interactions (i.e., solving conflicts) adhering to the principle that maximises social welfare, rather than individual utility. Although we present our conflict resolution approach in the setting of (a simplified version of) our reflective middleware, the problem we are addressing is general and applies to the design of applications that execute in highly dynamic settings, where adaptation, auto-configuration or self-healing become mandatory to achieve reasonable quality-of-service.

The remainder of the paper is structured as follows. In Section 2, we indicate the origins of this research and discuss our position compared to related work. In Section 3 we describe the core characteristics of our reflective mobile middleware and in Section 4 we provide a classification of the types of conflict that may arise in a mobile setting. Section 5 formalises the microeconomic mechanism we propose to solve these conflicts and Section 6 provides a comprehensive example that clarifies how our approach effectively works. Section 7 describes and evaluates our current implementation, and, finally, Section 8 concludes the paper and identifies some future work.

2. RELATED WORK

The problem of resolving conflicts is a general one and different research strands have investigated it over the years.

The operating systems community has studied the issue of conflicts in a distributed environment, where conflicts manifest as processes competing for shared resources. Microeconomic techniques, and auctions in particular, have been explored; in [15], 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 [21] and allocation of storage space [10]. We assess that game theory [1] tools can be successfully used to resolve conflicts that arise in the mobile setting too; however, rather than dealing with resource conflicts, we are interested into quality-of-service conflicts in service provision.

Despite the extensive research that has been carried out within the mobile middleware community, the issue of 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 [14, 2] 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.

The software engineering community has investigated the issue of conflicts too. Software development environments [9, 7] 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 [6] 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 [22] can be applied so that requirement conflicts are not come down to design. Other requirements engineering approaches [13] 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 [11] 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 [19] 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 [1] 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., [25] [24]). 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 to achieve it; as a consequence, negotiation mechanisms are usually iterative processes which proceed until an (optimal) agreement is found. In a mobile setting, instead, resource constraints call for simple conflict resolution mechanisms that do not waste the (scarce) resources applications need to achieve their goals. 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 middleware the quality of service levels that applications are willing to achieve.

Also relevant to our work is the research on quality of service provision in a mobile computing environment [5]. 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 [3]. 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.

In this paper, we show how middleware can use microeconomic techniques effectively in order to solve conflicts that arise in the mobile setting, where new issues (e.g., dynamic adaptation to context changes) and constraints (e.g., resource-scarce devices and low-quality network connection) have to be considered.

3. THE REFLECTIVE MODEL

The reflective middleware model we have developed (see [4] for a full discussion) assumes a single user for each mobile device, though there may be many applications running simultaneously on that device, hence, on the same middleware instance. This assumption is reasonable for PDAs, mobile phones and digital cameras. Applications need to be aware of their execution context, in order to adapt to frequent and unannounced changes in the environment. By context, we mean everything that can influence the behaviour of an application, from resources internal to 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. The middleware is in charge of maintaining a valid representation of the execution context, by directly interacting with the underlying network operating system.

Applications may require some services to be delivered in different ways (using different policies) when requested in different context. For example, an image processing application may wish to display pictures in black and white when battery power is low, using full-size, full-colour pictures when battery power permits. In our model, the middleware provides applications with a general mechanism that enables dynamic customisation of service delivery. The customisation takes place by means of what we call *application profiles*; Figure 1 shows their abstract syntax. Each application profile defines 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 service 'Display Picture', the 'Black&White' policy, and a context where the resource 'Battery Power' is low, and another one between the same service 'Display Picture', the 'FullColor' policy, and a context where 'Battery Power' is high. Profiles are passed down to the middleware; each time a service is invoked, the middleware consults the profile of the application that requests it to

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

Figure 1: Application profile's abstract syntax. $sname \in \mathrm{S}$, $pname \in \mathrm{P}$, $rname \in \mathrm{R}$, **being** $\mathrm{S}, \mathrm{P}, \mathrm{R} \subset \Sigma^*$**, respectively, the sets of all valid service/policy/resource names over our alphabet** Σ. $value \in \mathrm{V}$, **being** V **the set of all possible values of resources in** R **(e.g., IP addresses for hosts in reach, etc.);** $oname \in \mathrm{O}$, **being** O **the set of all valid operator names that can be applied to values of monitorable resources (e.g.,** ***equals*****,** ***lessThan*****).**

determine which policy can be applied in the current context. Our model assumes that, at each time, the behaviour of the middleware with respect to a particular service is determined by one and only one policy, that is, a service cannot be delivered using a combination of different policies. If different policies need to be combined, a new name must be assigned to the combined policy, and this name must be used in the profile. For example, the display of an image can be done using a 'B&W-Low' policy, that is a combination of 'Black&White' and 'LowResolution'.

As both the needs of the user 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. Here is where reflection comes into play. A reflective API is available that gives applications dynamic access to their own profile, so that changes in this information immediately translate into changes in the middleware behaviour.

4. DEALING WITH CONFLICTS

The model presented above allows applications to control the behaviour of the middleware based on current context. This is achieved by means of application profiles that can be dynamically changed through reflection. 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 (recall 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 from them 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 kinds of conflict may be generated.

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. Let us assume that we are running the image processing application described in Section 3. The application may instruct the middleware to display an image in black and white when the battery power on the device is low, and to load and display only fragments of the picture when memory is low. What happens when, suddenly, both battery and memory fall below the values specified in the profile? The middleware checks which policy should be applied and determines that more than one policy suits the current context. As we made the assumption that each service is delivered using one and only one policy at a time, the middleware is stuck, unable to choose which of the context-suitable policies to apply. This is an example of *intra-profile conflict*.

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. As a particular example of inter-profile conflict, we consider the case in which a conflict arises between applications running on two different devices. This scenario is typical of a mobile setting, where interactions take place between peers. Let us assume that we are running an instance of a calendaring application on our PDA; when meeting a colleague who is running the same application on his/her PDA, we want to synchronise our diary entries. However, the application profiles on the two hosts may conflict in the following way. While one of the two application instances may wish to synchronise data with its peer bidirectionally, regardless of the current execution context, the other one may prefer to communicate its updates to the peer, without getting the peer updates back, when memory availability is scarce. If the two hosts meet when the memory available on the second device is less than the amount specified in the profile, they will not agree on which protocol to use to synchronise their data. We call this situation an *inter-profile conflict*. 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*.

None of these 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. A possible static approach would require us to check whether there is any intersection between the different contexts of the policies associated with 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 number of contexts 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. 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; that is, the middleware cannot check whether the new configuration is conflict-free.

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. When this happens, the middleware has to resolve the conflict using an appropriate mechanism. This mechanism must be *simple*, that is, only a low computation and communication overhead should be imposed, as hand-held devices have limited resources that cannot be wasted by our conflict resolution mechanism. Moreover, it must be *customisable*, that is, it must be possible for the applications to influence the conflict resolution mechanism, and determine which policy is applied and which others are discarded.

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

5. MICROECONOMIC MECHANISM

When applications that participate in the delivery of a service do not agree on which policy must be applied, a conflict resolution scheme is necessary to resolve the dispute. The conflict resolution mechanism we propose is based on microeconomic techniques. 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 'Black&White', 'FullColor' and 'LowResolution' are the goods associated to service 'DisplayPicture'. Consumers are applications seeking to achieve their own goals, that is, to have the middleware delivering a service using the policy that achieves 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 [16]. A *protocol*, or mechanism, consists of a set of rules that govern interactions, and 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 contend that the auction protocol we have designed can be successfully applied in a mobile setting, where the requirements listed in Section 4 must be satisfied.

In the remainder of this section, we describe the details of the auctioning mechanism we propose to resolve conflicts in a mobile environment, and discuss why this mechanism achieves the requirements listed before.

5.1 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 application 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. This is due to the fact that we are considering scenarios where 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 on average 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 [23]). Unlike ascending bid auctions, such as the standard English auction [18], where the auctioneer 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 5.2 how customisation is met by our auctioning mechanism.

In the following, we provide the details of how our auctioning mechanism works in a mobile distributed setting. 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'[1]. Peers are partners in the communication process. We call PEERS the set of all possible peers. Under these assumptions, the auctioning process can be formalised as follows.

[1] We do not refer here to the characteristics of peers in P2P networks.

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 functions to find out how much to 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 its 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 asks the middleware to execute a service, a command like the one illustrated below is issued:

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

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 follow:

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

being $\mathcal{F}$ the semantic function defined in Figure 2; $serv : \mathrm{S} \times \mathrm{PEER} \rightarrow service$ 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:

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

being $\mathcal{I}$ the semantic function described in Figure 3.

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 Bets
For every peer $peer_i$ participating in the communication process, and for every agreed policy $p_j \in P^*$, $j \in [1, m]$, a bet $b_{i,j}$ is computed, based on the peer's 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 valued the use of each agreed policy:

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

being $\mathcal{B}$ the semantic function shown in Figure 4.

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:

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

being $\mathcal{W}$ the semantic function defined in Figure 5. As shown, before policy $p_{\bar{\jmath}}$ is actually applied, each peer 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 payment is split, let us consider three peers x, y and z, who bid respectively $b_x < b_y < b_z$ 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 respectively 0, $(b_y - b_x)/2$, and $(b_y - b_x)/2 + (b_z - b_y)/1$. Note that, if the winning policy is the one that has been valued most by all peers, 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 that requires the cooperation of a set of peer $peerList$, then the whole conflict resolution mechanism can be summarised as follows:

$$\begin{array}{l} \mathcal{G} : command \rightarrow \mathrm{P} \\ \mathcal{G}[|sname\ peerList|] = \\ \qquad \mathcal{W}\ [|\ \mathcal{B}\ [|\ \mathcal{I}[|sname|]_{\{peerList\}}\ |]_{\{peerList\}}\ |] \end{array}$$

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 valued most during a conflict resolution process);
- $\mathcal{G}[|sname\ peerList|] = \epsilon$: the service request fails as no policy can be found that is both agreed 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., 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 $peer_1$'s application profile). The auction proceeds as described above, selecting the policy that maximises $peer_1$'s utility, without communication costs.

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

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

Figure 2: Application profile semantics. $\mathrm{E} = \wp(\mathrm{R} \times \mathrm{V})$ 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 $rname$ in the execution context e is among the values obtained by applying the operator $oname$ to $valueList$ (e.g., $eval((Memory, inBetween, [2, 7]), \{(Memory, 6)\}) = \top$, while $eval((Memory, lessThan, [5]), \{(Memory, 6)\}) = \bot$).

$$
\begin{array}{rcl}
\mathcal{I} & : & S \rightarrow \wp(\mathrm{PEER}) \rightarrow \wp(\mathrm{P}) \\
\mathcal{I}[|sname|]_{\{peer\ peerList\}} & = & \mathcal{I}[|sname|]_{\{peer\}} \cap \mathcal{I}[|sname|]_{\{peerList\}} \\
\mathcal{I}[|sname|]_{\{peer\}} & = & \mathcal{F}[|serv(sname, peer)|]_{\mathcal{E}nv(peer)}
\end{array}
$$

Figure 3: Semantics of the computation of the solution set

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

Figure 4: Semantics of the computation of bets

$$
\begin{array}{rcl}
\mathcal{W} & : & \wp(\mathrm{P} \times \mathrm{PEER} \times \mathbb{R}^+) \rightarrow \mathrm{P} \\
\mathcal{W}[|\{(p_j, peer_i, b_{i,j}),\ \forall i \in [1,n], j \in [1,m]\}|] & = & p_{\tilde{j}} \mid \\
 & & p_{\tilde{j}} \in \{\pi_1(p_j, peer_i, b_{i,j}),\ \forall i \in [1,n], j \in [1,m]\} \\
 & & \wedge \sum_{i=1}^{n} \pi_3(p_{\tilde{j}}, peer_i, b_{i,\tilde{j}}) = \max_{j \in [1,m]} \sum_{i=1}^{n} \pi_3(p_j, peer_i, b_{i,j}) \\
 & & \wedge\ pay(q_{mw}(i), f_i, q_i),\ \forall i \in [1,n] \\
 & & \text{where } f_i = \begin{cases} \text{a. } 0 \text{ if } \forall k \in [1,n]\ \pi_3(p_{\tilde{j}}, peer_k, b_{k,\tilde{j}}) = \max_{j \in [1,m]} \pi_3(p_j, peer_k, b_{k,j}) \\[2ex] \text{b. } \sum_{\substack{l \in \{s \mid s \in [1,n] \\ \wedge b_{s,\tilde{j}} \leq b_{i,\tilde{j}}\}}} \frac{b_{l,\tilde{j}} - \max(\ \{b_{s,\tilde{j}} \mid b_{s,\tilde{j}} < b_{l,\tilde{j}},\ s \in [1,n]\} \cup \{b_{min,\tilde{j}}\}\ )}{\#\{b_{s,\tilde{j}} \mid b_{s,\tilde{j}} \geq b_{l,\tilde{j}},\ s \in [1,n]\} * \#\{b_{s,\tilde{j}} \mid b_{s,\tilde{j}} = b_{l,\tilde{j}},\ s \in [1,n]\}}, \\ b_{min,\tilde{j}} = \min\{b_{i,\tilde{j}}, i \in [1,n]\} \text{ otherwise} \end{cases} \\
\mathcal{W}[|\{(p,\ \_\,, 0)\}|] & = & p \ \text{No conflict} \\
\mathcal{W}[|\emptyset|] & = & \epsilon \ \text{No agreement}
\end{array}
$$

Figure 5: Semantics of the election of the winning policy. $\pi_i(a_1, a_2, \ldots, a_n) = a_i$ is a function that projects a tuple onto the i^{th} value; $\#\{a_1, a_2, \ldots, a_n\} = n$ is a function that computes the cardinality of a set; $q_{mw}(i)$ is a function that retrieves the quota of the middleware on top of which peer $peer_i$ is executing; finally, $pay(q_1, x, q_2) = (q_1 + x, q_2 - x)$ is a function that both increases the middleware quota q_1, and decreases the peer's quota q_2, of the specified amount.

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 not usually 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 on in isolation: a peer cannot expect that its behaviour will in any way affect a future conflict or, similarly, that it will behave in a particular way based on its interaction history (i.e., 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). Each conflict resolution process stands therefore alone.

There are some 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.

5.2 Utility Function

Whenever a conflict is detected, either inside a profile (intra-profile conflict) or among various profiles (inter-profile conflict), user's goals, such as maximisation of the quality of the display for an image processing application, or directionality of data synchronisation for a calendaring application, 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's 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 have 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 change *dynamically* to reflect changes in the user's goals; however, the value they return is computed over *static* policy specifications that 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 $level ::= \; '1'|\ldots|'\mathrm{LMAX}'$ an estimate of resource consumption/benefit achieved that the policy developers compute before delivering the policy.

The abstract syntax of a utility function is given in Figure 6, where $cb\_name \in (\mathrm{R} \cup \mathrm{Q})$ is a name that uniquely identifies a resource or benefit inside a policy specification, and $weight ::= '1'|\ldots|'\mathrm{WMAX}'$ is a factor that represents the importance the user gives to a particular resource/benefit (the higher the weight, the more important the resource/benefit). In this paper, we do not discuss how weights, that represent user's needs as faithfully as possible, can be generated, as we consider this issue a matter of future research.

Whenever a peer $peer_i$ is involved in a bidding process, its utility function is retrieved and used to find the peer's evaluation $u_{i,j}$ for each conflicting policy p_j. The semantics of a utility function is presented in Figure 7. As shown, each value is normalised to vary in a range $[0,1]$, so that different bets can be effectively compared, and money fairly redistributed (see Section 5.3).

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

Figure 6: Utility function abstract syntax

As stated before, while policy specifications are fixed, utility function specifications change over time, as they have to reflect current user's needs. This implies that the reflective API of our middleware (see Section 3) has to allow dynamic modification of utility function specifications. This allows our conflict resolution scheme to achieve also the second requirement we aimed to, 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.

5.3 Quota Allocation

When describing the rules of our mechanism (see Section 5.1), 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 in a bidding process, its current quota is decreased by an amount equal to $f_i \in [0,1]$, as defined in Figure 5. A_i's underlying middleware instance collects A_i's 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). 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 during the last t time units, then the recharging process is carried out as described below:

$$\begin{aligned} q_i &= q_i + \left(\overline{q}(i) - \frac{\overline{q}(i)}{N_t(i)}\right) \\ \overline{q}(i) &= \frac{\overline{q}(i)}{N_t(i)} \end{aligned}$$

being $\overline{q}(i)$ the money currently stored by the middleware in the

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

Figure 7: Semantics of an utility function. $\mathcal{S} : (\mathrm{R} \cup \mathrm{Q}) \rightarrow \mathrm{PSPEC} \rightarrow level$ **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, we consider a value of** 0**).** $intval$ **is a function that given a literal in** $\{'1', \ldots, 'MAX'\}$**, returns the corresponding integer value in** $[1, \mathrm{MAX}]$**;** $LMAX * WMAX * RQMAX$ **is the maximum bid an application can place, being** $RQMAX$ **the maximum number of resources/benefits of interest to an application.**

wallet associated to A_i, and q_i equals to A_i's 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 applications 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 8).

The approach to quota redistribution that we have described could be defined as 'conservative': at any time, the money associated to an application instance A_i are the same, 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 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 demonstrate an optimal choice for q_{init}, q_{max}, t and α.

Time / Action	$\mathbf{q_1}$	$\overline{\mathbf{q}}(\mathbf{1})$	$\mathbf{q_2}$	$\overline{\mathbf{q}}(\mathbf{2})$
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

Figure 8: Example of quota redistribution (with $t_7 - t_0 = t$**)**

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.

6. EXAMPLE

In this section, we present an example of inter-profile conflict using an instant messaging application on top of our reflective middleware, and show how our auctioning mechanism can be successfully applied to resolve it.

One of the services that our instant messaging application can customise is the way messages are delivered among peers that have started a chat. In particular, our middleware provides four different policies among which the application can choose: a `SendChar` policy, which is used to deliver to the peers a character at a time; a `SendLine` policy, which is used to deliver to the peers a line of characters at a time; a `SendZippedLine` policy, which first compresses a line of characters and then sends it to the peers, and finally a `SendEncryptedLine` policy which first encrypts and then sends a line of characters to the peers. Each of these policies requires a different amount of resources to be used (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 9.

```
SendChar
{(Battery,4),(Bandwidth,10),(Availability,10)}
SendLine
{(Battery,3),(Bandwidth,6),(Availability,7)}
SendZippedLine
{(Battery,5),(Bandwidth,4),(Availability,5)}
SendEncryptedLine
{(Battery,6),(Bandwidth,7),(Availability,4),(Privacy,10)}
```

Figure 9: Policy specifications

Let us suppose that three peers $peer_1$, $peer_2$, and $peer_3$ get in reach of each other and want to start a chat. In order to do so, we assume 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 10. The first peer enables each of the four policies in different contexts; the second peer prevents the use of the two heaviest policies, `SendChar` and `SendEncryptedLine`; finally the third one prevents the use of `SendChar`, while leaving `SendLine` always enabled (there is in fact no context associated to it). Leaving one or more

```
% peer 1
SendMessage
    SendChar
        Bandwidth > x1
    SendLine
        Bandwidth < x1
    SendZippedLine
        Bandwidth < x1/2
    SendEncryptedLine
        Battery > y1

% peer 2
SendMessage
    SendLine
        Battery < x2
    SendZippedLine
        Bandwidth < y2
```

```
% peer 3
SendMessage
    SendLine

    SendZippedLine
        Bandwidth < x3
    SendEncryptedLine
        Battery > y3
```

Figure 10: Application profiles

```
% peer 1            % peer 2           % peer 3
Battery         4   Battery      7     Privacy  10
Bandwidth       3   Bandwidth    9
Availability   10
```

Figure 11: Utility function specifications. $peer_1$ aims at maximising availability without wasting to many resources; $peer_2$ aims at minimising resource consumption, and $peer_3$ aims at maximising privacy.

policies always enabled 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, therefore, the time used to resolve them (which is anyway rather low, as it will be shown in the next section). It is up to the application to decide which strategy is best.

Assuming that the utility functions are the ones shown in Figure 11, and that the current execution context enables the following sets of policies:

$$P_1 = \{\texttt{SendLine}, \texttt{SendZippedLine}, \texttt{SendEncryptedLine}\}$$
$$P_2 = \{\texttt{SendLine}, \texttt{SendZippedLine}\}$$
$$P_3 = \{\texttt{SendLine}, \texttt{SendZippedLine}, \texttt{SendEncryptedLine}\}$$

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

Computation of the solution set

$$\mathcal{I}[|\texttt{SendMessage}|]_{\{peer_1, peer_2, peer_3\}} = P_1 \cap P_2 \cap P_3$$
$$= \{\texttt{SendLine}, \texttt{SendZippedLine}\}$$

Computation of bets

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: $\mathrm{LMAX}-$ expected consumption. For example, if we assume $\mathrm{LMAX} = 10$, $\mathrm{WMAX} = 10$, and $\mathrm{RQMAX} = 4$ (i.e., battery, bandwidth, availability and privacy), then:

$$\begin{aligned} u_{peer_1}(\texttt{SendLine}) &= \frac{(10-3)*4 + (10-6)*3 + 7*10}{10*10*4} \\ &= 110/400 = 0.275 \end{aligned}$$

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

$$\begin{aligned} &\mathcal{B}[|\{\texttt{SendLine}, \texttt{SendZippedLine}\}|]_{\{peer_1, peer_2, peer_3\}} = \\ &\{(\texttt{SendLine}, peer_1, 0.275), (\texttt{SendZippedLine}, peer_1, 0.22), \\ &(\texttt{SendLine}, peer_2, 0.2125), (\texttt{SendZippedLine}, peer_2, 0.2225), \\ &(\texttt{SendLine}, peer_3, 0), (\texttt{SendZippedLine}, peer_3, 0)\ \} \end{aligned}$$

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:

$$\begin{array}{rcccc} & peer_1 & peer_2 & peer_3 & \\ \texttt{SendLine}: & 0.275 + & 0.2125 + & 0 = & 0.4875 \\ \texttt{SendZippedLine}: & 0.22 + & 0.2225 + & 0 = & 0.4425 \end{array}$$

Therefore:

$$\mathcal{W}[|\ \mathcal{B}[|\{\texttt{SendLine}, \texttt{SendZippedLine}\}|]_{\{peer_1, peer_2, peer_3\}}|] = \texttt{SendLine}$$

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

$$\begin{aligned} q_1 &= q_1 - \frac{0.275 - 0.2125}{1} - \frac{0.2125 - 0}{2} - \frac{0}{3} \\ q_2 &= q_2 - \frac{0.2125 - 0}{2} - \frac{0}{3} \\ q_3 &= q_3 - \frac{0}{3} \end{aligned}$$

7. IMPLEMENTATION AND EVALUATION

We have implemented our middleware in Java that supports the full reflective model and the conflict resolution mechanism described in this paper. We have also developed an instant messaging application on top of it and ran tests on a Compaq iPAQ PDA running Linux. Application profiles and utility functions have been encoded using the eXtensible Markup Language (XML). 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 is both easily manipulatable by machines and readily understandable by humans.

The middleware platform currently requires less than 250Kb of persistent storage, and less than 800Kb of memory (without considering the memory required by the Java Virtual Machine). We have measured the average time required to start a service, with and without conflicts. We have experienced no time increase in the case of intra-profile conflicts, and an increase of around 20% (450ms instead of 365ms) in case inter-profile conflicts between two peers were detected and solved (with only 300 bytes of information sent around). In both cases, there was no human perception of the delay as most of the time is taken by the loading and provisioning of the service.

In our auctioning scheme, the role of the auctioneer can be played by more than a middleware instance at the same time, as bidders may be distributed across different machines (inter-profile conflicts); this requires cooperation among all the middleware instances involved, to compute the solution set and to exchange bids. The communication paradigm we are currently using requires that all the peers remain connected while the conflict resolution mechanisms is performed, otherwise a failure is reported. We plan to investigate and develop different communication paradigms that allow our auctioning protocol to be resilient even in the presence of individual's disconnections.

8. CONCLUSION AND FUTURE WORK

In this paper we presented a microeconomic approach to conflict resolution in a mobile setting. In particular, we 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, middleware 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 simplicity and customisability that are typical of this environment.

The auction protocol we used can be extended in many directions. For example, we based our mechanism on the assumption that each service is delivered using one policy at a time; however, this policy may be the result of applying a set of other (sub)policies one after the other. Considering the example illustrated in Section 6, policy `SendZippedLine` is a combination of `Zip` and `SendLine`. Our mechanism could be extended to deal with conflicting situations where middleware is in charge of both selecting a winning policy, and fairly distributing the various (sub)policies to the participating applications. Applications could be asked, for example, to bid a price, not only for a policy, but also for its constituents. Moreover, if applications do not agree on a common set of enabled policies (i.e., the solution set is empty), the conflict resolution process fails. To reduce the risk of such failures, our auctioning mechanism could be extended so that applications that weaken their requirements, and enable a larger set of policies, could be rewarded (e.g., through quota recharging). However, the cost of these approaches should be carefully considered as we are targeting a mobile setting characterised by resource-scarce devices, and we do not want to compromise efficiency while handling the conflict resolution process.

Our plans for the future include an investigation of how user's needs can be dynamically mapped into application profiles and utility functions. In [6], a goal-oriented approach has been investigated to operationalise user's goals into services of the system. In [12], a reflective extension of this work has been proposed in order to allow system behaviour to adapt to the changing context (as it is necessary in a mobile environment). In order for this to be feasible, the context (and its changes) must be represented at run-time. This representation must take place in a way that is both readily understandable by humans and easily manipulatable by machines. Our work has already moved in this direction, with the idea of using XML to represent both context and profiles; we plan to investigate this issue further.

Acknowledgements. We would like to thank 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; Anthony Finkelstein and Bashar Nuseibeh for their comments on an earlier draft. Finally, we thank the anonymous reviewers, who have produced detailed reviews and much helped to improve this paper.

9. REFERENCES

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

[2] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proc. of Middleware '98*, LNCS, pages 191–206. Springer Verlag, Sept. 1998.

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

[4] 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*, volume 2192 of *LNCS*, pages 126–133, Kyoto, Japan, Sept. 2001.

[5] D. Chalmers and M. Sloman. A Survey of Quality of Service in Mobile Computing Environments. *IEEE Communications Surveys*, 2(2), 1999.

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

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

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

[9] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[10] 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*, pages 367–375, Los Alamitos, CA, 1993. IEEE Computer Society Press.

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

[12] A. Finkelstein and A. Savigni. A Framework for Requirements Engineering for Context-Aware Services. In *Proc. of 1^{st} International Workshop From Software Requirements to Architectures (STRAW 01)*, Toronto, Canada, May 2001.

[13] A. Hunter and B. Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Trans. on Software Engineering and Methodology*, 7(4):335–367, Oct. 1998.

[14] T. Ledoux. OpenCorba: a Reflective Open Broker. In *Reflection'99*, volume 1616 of *LNCS*, pages 197–214, Saint-Malo, France, 1999. Springer.

[15] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In B. A. Huberman, editor, *The Ecology of Computation*, pages 177–205. North-Holland, Amsterdam, 1988.

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

[17] C. Mascolo, L. Capra, and W. Emmerich. Middleware for Mobile Computing (A Survey) . In *Networking 2002 Tutorial Papers*, volume 2497 of *LNCS*. Springer, 2002.

[18] P. Milgrom. Auctions and Bidding: A Primer. *Journal of Economic Perspectives*, 3(3):3–22, 1989.

[19] W. N. Robinson and S. D. Pawlowski. Managing requirements inconsistency with development goal monitors. *IEEE Transactions on Software Engineering*, 25(6):816–835, 1999.

[20] G.-C. Roman, A. L. Murphy, and G. P. Picco. Software Engineering for Mobility: A Roadmap. In *The Future of Software Engineering - 22^{nd} Int. Conf. on Software Engineering (ICSE2000)*, pages 243–258. ACM Press, May 2000.

[21] 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.

[22] A. van Lamsweerde, R. Darimont, and E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, Nov. 1998.

[23] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16(1):8–37, 1961.

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

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