Wireless Personal Communications **21:** 77–103, 2002.
© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

XMIDDLE: A Data-Sharing Middleware for Mobile Computing

CECILIA MASCOLO, LICIA CAPRA, STEFANOS ZACHARIADIS and WOLFGANG EMMERICH
Department of Computer Science, University College London, Gower Street, London WC1E 6 BT, U.K.
E-mail: {C.Maxolo}{L.Capra}{S.Zachariadis}{W.Emmerich}@cs.ucl.ac.uk

Abstract. An increasing number of distributed applications will be written for mobile hosts, such as laptop computers, third generation mobile phones, personal digital assistants, watches and the like. Application engineers have to deal with a new set of problems caused by mobility, such as low bandwidth, context changes or loss of connectivity. During disconnection, users will typically update local replicas of shared data independently from each other. The resulting inconsistent replicas need to be reconciled upon re-connection. To support building mobile applications that use both replication and reconciliation over ad-hoc networks, we have designed XMIDDLE, a mobile computing middleware. In this paper we describe XMIDDLE and show how it uses reflection capabilities to allow application engineers to influence replication and reconciliation techniques. XMIDDLE enables the transparent sharing of XML documents across heterogeneous mobile hosts, allowing on-line and off-line access to data. We describe XMIDDLE using a collaborative e-shopping case study on mobile clients.

Keywords: mobile computing, middleware, data reconciliation, XML.

1. Introduction

According to Mark Squires (Nokia) it took 15 years for the TV to reach a cr itical mass of 50 million users, but it took the mobile phone industry only 18 months to sell 50 million phones in Europe alone. Mobile phones become increasingly computationally powerful, are integrated with PDA capabilities (e.g., Nokia's 9210) and are equipped with ad-hoc networking technologies (e.g., Ericsson's T36 that implements Bluetooth (Mettala, 1999)). Conversely, PDA's gain increasingly powerful wireless networking capabilities, by incorporating IrDA, Bluetooth, or 802.11b (WaveLan) hardware. For example, Xircom already ships a 802.11b module for the Handspring Visor series of PDAs, Palm has repeatedly expressed interest on the Bluetooth technology, implementing a prototype PDA with Bluetooth integrated. Moreover, Symbol announced a 802.11b Compact Flash card, giving wireless connectivity to PocketPC PDAs and laptop computers. Of further interest is that Anycom has announced the availability of a Compact Flash Bluetooth card for PDAs and laptop computers. Such capabilities enable new classes of applications to exploit, for example, the ability to form ad-hoc workgroups, but they also present new challenges to the mobile application developer. In particular, resources, such as available main memory, persistent storage, CPU speed and battery power are scarce and need to be exploited efficiently. Moreover, network connectivity may be interrupted instantaneously and network bandwidth will remain by orders of magnitude lower than in wired networks.

In distributed systems, the complexity introduced through distribution is made transparent to the application programmer by means of middleware technologies, which raise the level of abstraction. Existing middleware technologies, such as remote procedure call systems, dis-

tributed object middleware (Emmerich, 2000), and message- or transaction-oriented systems hide the complexities of distribution and heterogeneity from application programmers and thus support them in constructing and maintaining applications efficiently and cost-effectively. However, these technologies have been built for wired networks and are unsuitable for a mobile setting (Capra et al., 2001). In particular, the interaction primitives, such as remote procedure calls, object requests, remote method invocations or distributed transactions that are supported by current middleware paradigms assume a high-bandwidth connection of the components, as well as their constant availability. In mobile systems, instead, unreachability and low bandwidth are the norm rather than an exception. In Bayou (Petersen et al., 1997) disconnection was contemplated as a rare and occasional event. The system hides mobility from the application layer in the same way as transparency for relocation of object is used in modern middleware systems.

We rather believe that middleware systems for mobile computing need to find different kinds of interaction primitives to accommodate the possibility for mobile components to become unreachable. Many PDA applications copy, for example, agendas, to-do lists and address records from a desktop machine into their local memory so that they can be accessed when the desktop is unreachable. In general, mobile applications must be able to replicate information in order to access them off-line. Replication causes the need for synchronization when a connection is re-established. This need is not properly addressed by existing middleware systems. The commonly used principle of transparency prevents the middleware to exploit knowledge that only the application has, such as which portion of data to replicate and which reconciliation policy to apply. It seems therefore necessary to design a new generation of middleware systems, which disclose information previously hidden, in order to make best use of the resources available, such as local memory and network bandwidth.

Tuple space coordination primitives, that were initially suggested for Linda (Gelernter, 1985), have been used to facilitate component interaction for mobile systems. Tuple spaces achieve a decoupling between interacting components in both time and space by matching the idea of asynchronicity with the mobile computing embedded concept of disconnection and reconnection. Tuple spaces do not impose any data structures for coordination allowing more flexibility in the range of data that can be handled. On the other hand, the lack of any data structuring primitives complicates the construction of applications that need to exchange highly structured data.

In this paper we present XMIDDLE, which advances mobile computing middleware approaches by firstly choosing a more powerful underlying data structure and secondly by supporting replication and reconciliation. XMIDDLE's data structure are trees rather than tuple spaces. More precisely, XMIDDLE uses the eXtended Markup Language (XML) (Bray et al., 1998) to represent information and uses XML standards, most notably the Document Object Model (DOM) (Apparao et al., 1998) to support the manipulation of its data. This means that XMIDDLE data can be represented in a hierarchical structure rather than, for instance, in a flat tuple space. The structure is typed and the types are defined in an XML Document Type Definition or Schema (Fallside, 2000). XMIDDLE applications use XML Parsers to validate that the tree structures actually conform to these types. The introduction of hierarchies also facilitates the coordination between mobile hosts at different levels of granularity as XMIDDLE supports sharing of subtrees. Furthermore, representing mobile data structures in XML enables seamless integratin of XMIDDLE applications with the Micro Browsers, such as WML browsers in mobile phones, that future mobile hosts will include.

The paper is organized as follows: in Section 2 we briefly introduce XMIDDLE and the main characteristics of the system. XMIDDLE makes extensive use of XML and we sketch how we use XML and related technologies in Section 3. Section 4 introduces a case study that we use both to demonstrate and to evaluate the XMIDDLE concepts. XMIDDLE uses versioning to manage updates of replicas and we discuss the underlying versioning principles in Section 5. Section 6 contains the details of the protocols that we use for reconciliation and conflict resolution, tree linking and disconnection. Section 7 discusses the basic architecture of XMIDDLE and presents the primitives that this architecture provides for mobile applications. The section also describes our implementation prototype. In Section 8 we discuss and evaluate the XMIDDLE system and in Section 9 we conclude the paper and list some future work.

2. An Outline of XMIDDLE

XMIDDLE allows mobile hosts (i.e., PDAs, mobile phones, laptop computers or other wireless devices) to be physically mobile, while yet communicating and sharing information with other hosts. We do not assume the existence of any fixed network infrastructure underneath. Mobile hosts may come and go, allowing complicated *ad-hoc network* configurations. Connection is symmetric but not transitive as it depends on distance; for instance host $\mathtt{H}_A$ can be connected to host $\mathtt{H}_B$, which is also connected to host $\mathtt{H}_C$. However, host $\mathtt{H}_A$ and host $\mathtt{H}_C$ may be not connected to each other. Mobile network technologies, such as Bluetooth (Mettala, 1999) facilitate these configurations with multiple so called *piconets* whose integration forms *scatternets* in Bluetooth. We do not consider any multi-hop scenarios where routing through mobile nodes is allowed, but it is in our agenda to investigate this issue further.

In order to allow mobile devices to store their data in a structured and useful way, we assume that each device stores its data in a tree structure. Trees allow sophisticated manipulations due to the different node levels, hierarchy among the nodes, and the relationships among the different elements which could be defined. XMIDDLE defines a set of primitives for tree manipulation, which applications can use to access and modify the data.

When hosts get in touch with each other they need to be able to communicate. XMIDDLE therefore provides an approach to sharing that allows on-line collaboration, off-line data manipulation, synchronization and application dependent data reconciliation. On each device, a set of possible access points for the owned data tree are defined so that other devices can link to these points to gain access to this information; essentially, the access points address branches of trees that can be modified and read by peers. In order to share data, a host needs to explicitly *link* to another host's tree. The concept of linking to a tree is similar to the mounting of network file systems in distributed operating systems to access and update information on a remote disk. Access oints to a host's tree are a set that we call *ExportLink*. Let us say that host H_i exports the branch A and that hosts H_j and host H_k link to it, expressing the wish to share this information with host H_i. The owner of the branch is still host H_i but the data in the branch can be modified and read by the three hosts. The way sharing and data replication and reconciliation is allowed in XMIDDLE depends, however, also on additional conditions related to the connection status among the hosts.

In order to share data, hosts need to be *connected*. Host $\mathtt{H}_A$ becomes *connected* with host $\mathtt{H}_B$ when it is "in reach" of it.[1] When two hosts are connected they can share and modify

[1] The specific definition of *in reach* depends on the network protocols and hardware devices used. Considering wireless LAN and Bluetooth *in reach* means in radio range.

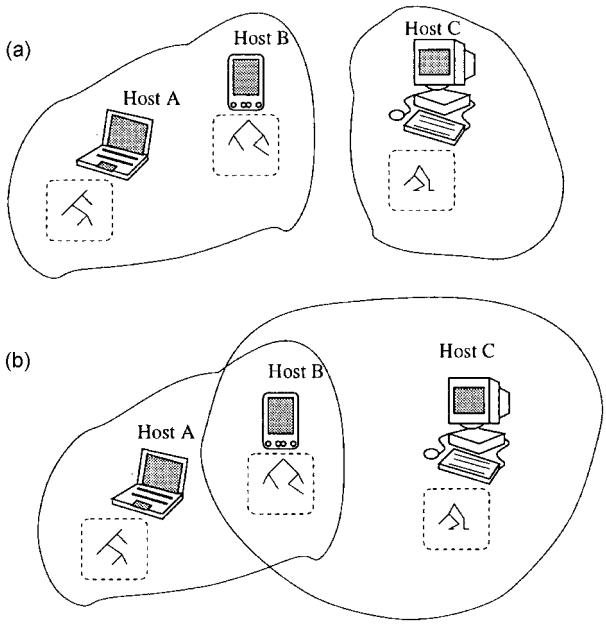

Figure 1. (a) Host $\mathtt{H}_B$ and Host $\mathtt{H}_C$ are not in reach. (b) Host $\mathtt{H}_B$ and Host $\mathtt{H}_C$ connect and Host $\mathtt{H}_B$ receives a copy of the XML tree that it has linked from Host $\mathtt{H}_C$.

the information on each other's linked data trees. Figure 1 shows the general structure of XMIDDLE and the way hosts get in touch and interact. Each host has full control over its own tree, however, it is obliged to notify other connected hosts that link to the modified part (*branch*) of its tree about the changes introduced. If, for instance, host H_k wishes to modify a branch A linked from the host H_i (owner of the branch), which is in reach, it requests H_i to perform the desired changes. H_i then notifies the changes to all the hosts (in reach) that link to the modified branch, including H_i.

The first time that the two hosts H_k and H_i are in reach of each others, the middleware on the hosts realizes that H_k is linking to the branch A of host H_i and a download process of the branch is started. Once downloaded the branch, host H_k may happen to go out of reach. The host is allowed to make off-line changes to branch A which will then be reconciled to the changes H_i did, when the two hosts get in reach again, if ever. While moving, H_k may happen to meet H_j, which is also linking to branch A of host H_i. Also in this case the reconciliation of the data takes place. A system of versions of the data in the tree is kept to allow this data reconciliation and sharing (Section 5).

A host records the branches that it links from other remote hosts in the set *LinkedFrom*, and the hosts linking to branches of the owned tree in the set *LinkedBy*. These sets contain lists of tuples (*host, branch*) that define the host that is linking to a branch, and from whom a branch is linked, respectively. *LinkedFrom* does not mirror the connection configuration, that is, host H_A can be in the *LinkedFrom* list of H_B also if the two hosts are not in reach (specific primitives for *linking* and *unlinking* trees modify these sets). On the contrary, the *LinkedBy* set is updated by *connection* and *disconnection* operations and it is used to know to whom to notify changes of parts of the tree.

Hosts may explicitly disconnect from other hosts, even though these hosts may be "in reacht". XMIDDLE supports explicit disconnection to enable, for instance, a host to save

battery power, to perform changes in isolation from other hosts and to not receive updates that other hosts broadcast. Disconnection may also occur due to movement of a host into an out of reach area, or to a fault. In both cases, the disconnected host retains replicas of the last version of the trees it was sharing with other hosts while connected and continues to be able to access and modify the data; the versioning system that we will describe later is in place to allow consistent sharing and data reconciliation.

3. XMIDDLE and XML

In the previous section we have described the motivation and main characteristics of XMIDDLE. We now give the details on how we use XML for structuring the device information as trees, and how XML related technologies are exploited in order to achieve linking and addressing.

XML documents can be semantically associated to trees. We therefore format the data located on the mobile devices as XML trees. The applications on the devices are enabled to manipulate the XML information through the DOM (Document Object Model) API (Apparao et al., 1998) which provides primitives for traversing, adding and deleting nodes to an XML tree. The implementation of this API, however, is XMIDDLE specific.

Furthermore, XML related technologies, and in particular XPath (Clark and DeRose, 1999), are used in XMIDDLE to format the addressing of points in a tree. *LinkedFrom*, *LinkedBy* and *ExportLink* sets are formatted using XPath. The XPath syntax is very similar to the Unix directory addressing notation. For instance, to address a node in an XML tree the notation used is `/root/child1/child2/`. We will give extensive example of use in the Case Study Section (Section 4).

The reconciliation of XML tree replicas which hosts use to concurrently and off-line modify the shared data, exploits the tree differencing techniques developed in (Tai, 1979). XMLTreeDiff (Alphaworks, 1998) is a package that implements this algorithm and that XMIDDLE uses to handle reconciliation. We note, however, that reconciliation cannot in all cases be completed by the XMIDDLE layer alone. Similarly to merging text files, tree updates may lead to differences which can be solved only using application-specific policies or may even need end-user interaction. The use of XML as an underlying data structure, however, enables XMIDDLE to both highlight the differences and define reconciliation policies specific to particular types of document elements, and therefore to specific applications (see Section 6).

4. A Case Study

In order to show how XMIDDLE supports building a mobile application we describe a collaborative electronic shopping system. Assme that a family has three members and that the family owns a PC and each member of the family has a PDA. Assume further that the PC and the PDAs have embedded Bluetooth technology to establish ad-hoc networks. The family does its weekly shopping electronically. To do so, the PC maintains a replica of the shop's product catalogue that is encoded in XML, as sketched in Figure 2. The catalogue on the PC is updated whenever a price or the portfolio of the shop changes. Family members replicate subsets of the product catalogue on their PDAs. We suppose that the different members hold replicas of different parts of the catalogue as they are interested in different product categories. For example, the mother may have an interest in beauty products, the father in hardware and the

```
<shop lastupdate="2001-01-21"/>
 <category name="Dairy">
  <product>
   <name> Milk</name>
   <price> 1.20</price>
  </product>
  <product>
   <name> Cheese</name>
   <price> 3.50</price>
  </product>
 </category>
 <category name="Fruit">
  <product>
   <name> Apple</name>
   <price> 2.20</price>
  </product>
  <product>
   <name> Pear</name>
   <price> 1.60</price>
  </product>
 </category>
</shop>
```

Figure 2. The XML representation of the product catalogue.

child in sweets and toys. The product categories however may overlap among the members. To show this in our example, we assume that Member_A is only interested in dairy products, Member_B in fruit, while Member_C is interested in both dairy and fruit. Furthermore, each family member has a replica of a joint shopping basket. They shop by dragging items of the catalogue into their shopping basket and by selecting quantities for these items. Reconciliation of product catalogues and shopping baskets happens whenever the PDAs establish connection to each other or to the PC.

Both the PC's catalogue subtrees and the PC's basket can be linked using the "link" operation provided by XMIDDLE. When a PDA gets within reach of the PC for the first time after the link operation, it reconciles the PDA's version with the PC's version by transferring catalogue subtrees and the empty basket to the PDA. Member_C, for example, may decide to link to the whole catalogue in addition to the empty basket. To link only to the dairy category on Member_A's PDA, it specifies the path of the DOM tree of that category and also links to the empty shopping basket. Figure 3 shows how Member_A and Member_C link to the categories (i.e., dairy products for Member_A and the whole catalogue for Member_C) and the empty basket, which the applications on their respective PDAs will fill with products to be purchased.

The first parameter of `link()` operation is the server host name, in this case the PC. The second parameter is the XPath expression (Clark and DeRose, 1999) for the root of the branch to be linked. Consider the linking expression for the "Dairy" products branch in Figure 3, for which we use a predicate XPath expression to select that category element, whose value of attribute `name` equals `Dairy`. The third parameter of `link()` operation is the "mounting point"

```
//MemberA's link requests
link("PC.home.net","/shop/category[@name="Dairy"],/);
link("PC.home.net","/basket",/);

//MembersC's link requests
link("PC.home.net", "/shop",/);
link("PC.home.net", "/basket",/);
```

Figure 3. Use of linking mechanism for Member$_A$ and Member$_C$.

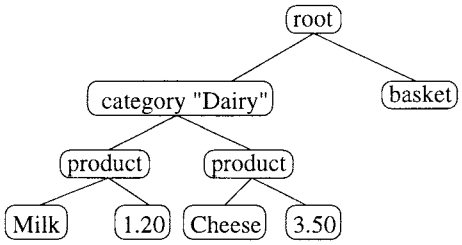

Figure 4. The tree representation on Member$_A$'s PDA.

on the local host. The resulting virtual XML trees containing the linked parts are shown in Figures 4 and 5.

The application on each PDA can now use DOM primitives to traverse the catalogue in order to display different categories and products. To implement the addition of new items to the shopping basket, DOM operations are used to create new child nodes of the shopping basket node. Let us suppose that Member$_A$ begins to put products into the basket; a sample configuration is shown in Figure 6, where an order for one bottle of milk has been added to the basket. If these orders are entered while the PDA is connected to the PC, the implementation of the DOM operations will request updates of the DOM tree from XMIDDLE middleware on the PC (as the PC is the "owner" of the branch). Let us now assume that the PDA was either out of reach of disconnected while these updates occurred.

When the PDA of a family member establishes a connection with the PC, the reconciliation protocol (details in Section 6) will reconcile any update that the PC has receied via the Internet from the shop. Likewise, any update that members have introduced to their shopping basket will be incorporated into the basket on the PC.

The PDA can also establish a connection with other PDAs when they meet in different rooms of the house or in town. The ability for every host to updte a replica opens the possibility of *conflicting* updates. As an example, let us now suppose that Member$_C$ is also buying milk

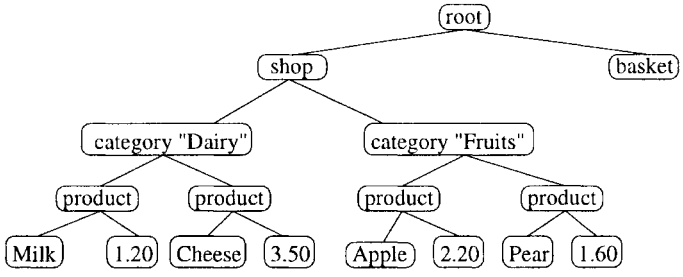

Figure 5. The tree representation on Member$_C$'s PDA.

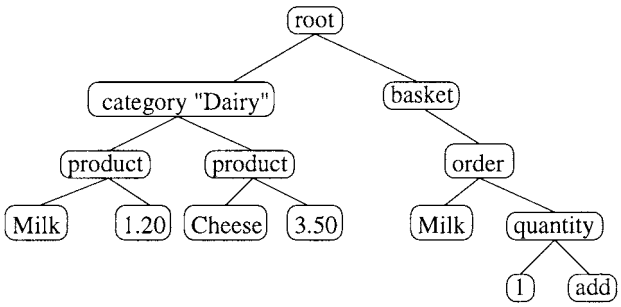

Figure 6. The tree representation of the data on Member$_A$'s PDA after an order has been entered.

from the dairy category, this time however ordering two bottles. When the two hosts Member$_A$ and Member$_C$ connect their PDAs, their XMIDDLE middleware realize they are both linking from the same host (i.e., the PC) common tree branches. The reconciliation process has to compute a consistent new version of the linked branches, both the basket and the shop. The PDAs may have two different versions of the shop catalogue if they synchronized with the PC at different times. The reconciliation of the catalogue among the PDAs allows the members to have a more up to date version of the catalogue even without connecting to the PC. The reconciliation of the basket allows a member to communicate his or her part of the shopping list to the other member so that, if one of them goes hom, this is immediately copied into the PC. Eventually (once a week), the home PC can then fire off the shopping list to the shop. The shopping basket on the PC is gradually filled through synchronization with the different members of the family.

We now focus on the basket differences: the reconciliation algorithm (which is described in Section 6 in detail) identifies that a *conflict* occurred as the quantities for the milk have different values in the two basket replicas. Unfortunately, we cannot resolve this conflict without using application-specific knowledge: only the application knows whether the total amount of bottles to be bought must be one (Member$_A$), two (Member$_C$) or three (sum of the two). We show in the following sections how XMIDDLE addresses this issue.

5. Versioning

Before giving the details of the reconciliation protocol, we explain formally how XMIDDLE manages different versions of DOM trees.

The principal data structure that XMIDDLE maintains for every host is a tree where each node contains a directed version graph of DOM trees from potentially different hosts (Figure 7). A version graph can contain two types of elements: *editions* and *versions*. Informally, an edition is a stable version that the host has agreed to save on persistent storage, e.g., in Flash RAM. We refer to the process of establishing a new edition as *releasing* a version. A version, on the contrary, is still subject to changes and it is only kept in main memory. This means that an edition can have both versions and editions as directed descendents in the version graph, while a version cannot have descendents at all: a version can only be derived from an edition. At the moment we assume that every host has no more than one open version of a tree, either linked or owned, and that this version has been created from the latest edition. We also assume that every host X is uniquely identified by an identifier $\mathtt{H}_X$.

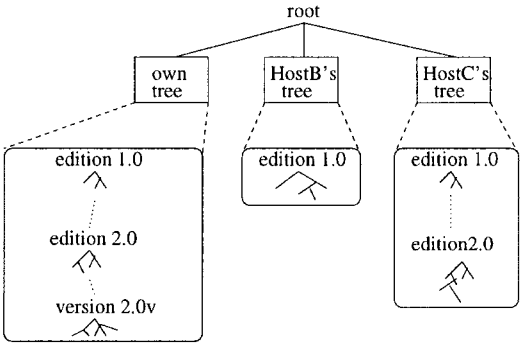

Figure 7. Tree of version history graph of DOM trees.

For every node in the `ExportLink` set, that is, for every remotely linkable point, XMIDDLE provides an *edition identifier* `EI` that uniquely identifies, in a distributed environment, an edition of the subtree with this node as root. This identifier is a tuple:

`EI(e_number, H`$_A$`, H`$_B$`),`

where H_A and H_B identify uniquely the at most two hosts[2] that agreed in releasing this edition. The edition number `e_number` is the increment of the maximum of the two previous edition numbers and it is used to disambiguate between subsequent editions agreed by the same couple of hosts. We assume that the sequence of edition numbers always starts from number 1. The edition number alone is not sufficient to distinguish among different editions. Distributions adds new complexity to the problem of versioning as we lack now a central authority to issue new edition numbers: it is possible for two hosts to reconcile a tree they copied from another host, without asking the owner, that is a central authority, for a new edition number. Let us consider for instance the scenario depicted in Figure 8: four hosts H_A, H_B, H_C and H_D have edition 1 of a tree `T` linked from host H_X. While disconnected, they modify their local version independently of each other; when H_A and H_B get in touch, they can reconcile this tree, creating a new edition with `e_number = 2`. The same can happen to H_C and H_D, leading to another (but different) edition 2. If now H_A and H_C connect and look only at the edition numbers they share, they may wrongly assume their latest common version of `T` is number 2. Our approach eliminates the problem as when H_A and H_C connect to each other, they recognize they have different versions of `T`, namely `EI(2, H`$_A$`, H`$_B$`)` and `EI(2, H`$_C$`, H`$_D$`)`; the only thing they can do now is to reconcile these different editions, generating `EI(3, H`$_A$`, H`$_C$`)`. A letter `v` attached to an edition number means that the corresponding node has been modified and that the changes have not been agreed yet; a symbol `$` for the host identifier means that the agreement did not involve a second host (H_A decided to create a new stable version without reconciling with anyone else).

The basic principle upon which our distributed versioning scheme relies on is the following.

[2] An edition can be created by a single host alone while disconnected or by two hosts as the final step of a reconciliation process. We assume that the reconciliation process is point-to-point, so no more than two hosts can be involved.

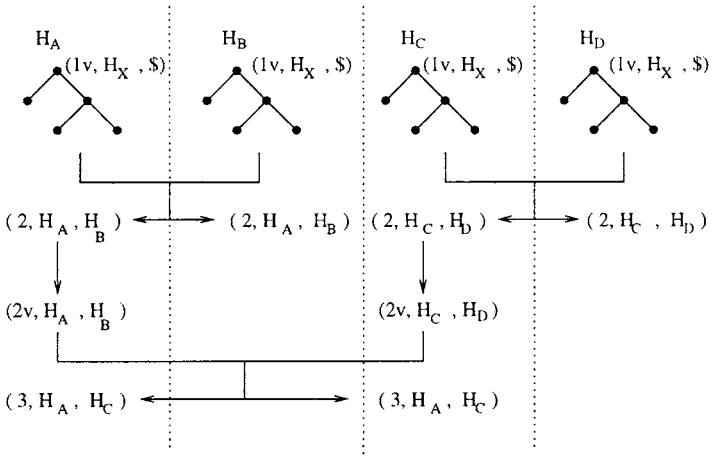

Figure 8. Uniqueness of edition identifiers `EI`.

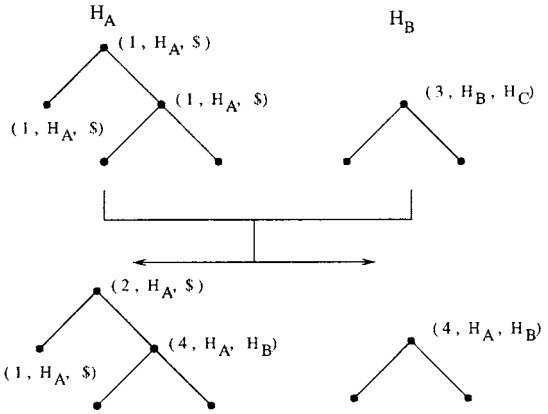

Figure 9. Distributed versioning scheme.

Versioning Principle
When releasing a subtree $\mathtt{T}'$ *of a tree* `T`, *for each changed node* $\mathtt{n} \in \mathtt{T}'$ *we increase the edition number of all the linkable nodes on the path from* `n` *included towards the root of* `T`. *When deriving a version from an edition, for every changed node* `n` *we mark the edition number of all the linkable nodes on the path from* `n` *included towards the root with a* `v`.

Figure 9 illustrates what happens to the edition numbers of the nodes of a tree linked by two different hosts. $\mathtt{H}_B$ has linked to a subtree $\mathtt{T}'$ of a tree `T` owned by $\mathtt{H}_A$. While disconnected from the owner, $\mathtt{H}_B$ has reconciled with $\mathtt{H}_C$, which is linking to the same subtree $\mathtt{T}'$. Once $\mathtt{H}_A$ and $\mathtt{H}_B$ get in reach again, they reconcile. As a result, a new edition identifier has been created for the root node of $\mathtt{T}'$ and, for the versioning principle described above, the same happens to the root node of `T`. This is necessary in order for other hosts moutning the whole tree `T` to realize that it has changed since the last time they reconciled with $\mathtt{H}_A$ (this will be clearer when describing the details of the reconciliation process, in the following section). Nothing happens to the left branch of `T` instead, as it has not been affected by any changes during the reconciliation process.

6. Protocols

In this section we describe the protocols that we use to *reconcile* the trees that two hosts share, to *link* a (sub)tree from one host to another, and to *disconnect* a host.

6.1. Reconciliation Protocol

The aim of reconciliation is to obtain a consistent version of the replicas of the same tree once two hosts become connected. Our reconciliation approach is composed of two main parts, one of which is application-independent and one application-specific. The former is based on general techniques for XML tree comparison and merging, while the latter allows us to tune reconciliation parameters for resolving conflicts in an application-specific way. We discuss the two parts separately.

Application-Independent Reconciliation
Without loss of generality, we assume that two hosts $\mathtt{H}_A$ and $\mathtt{H}_B$ get in reach after having worked off-line for a while on the same tree branch. The following reconciliation protocol is started. We use the symbol $\mathtt{X} \rightarrow_{\mathtt{msg}} \mathtt{Y}$ to mean that message `msg` has been sent from `X` to `Y`.

1. $\mathtt{H}_B \rightarrow_{\mathtt{LinkedFrom_B,ExportLink_B}} \mathtt{H}_A$
2. $\mathtt{H}_A \rightarrow_{\mathtt{LinkedFrom_A,ExportLink_A}} \mathtt{H}_B$
   Each time two hosts get in reach, they exchange their `LinkedFrom` and `ExportLink` sets, in order to see whether they share some information. When they realize they share a branch `T`, they first lock it and then start the actual reconciliation process. If one of the hosts is the owner of the branch `T`, it also flushes the queue of pending requests for changes (received from the on-line hosts linking that branch).
3. $\mathtt{H}_B \rightarrow_{\mathtt{TlistOfEI}} \mathtt{H}_A$
   $\mathtt{H}_B$ sends the list of all the edition identifiers for `T` starting from the latest one until the root of the version history graph to $\mathtt{H}_A$.
4. $\mathtt{H}_A \rightarrow_{\mathtt{lastSharedEI,listOfChanges}} \mathtt{H}_B$
   $\mathtt{H}_A$ determines the most recent common edition it shares with $\mathtt{H}_B$ (`lastSharedEI`).[3] $\mathtt{H}_A$ then computes the changes done sine then and sends this list of differences together with `lastSharedEI`, to $\mathtt{H}_B$.
5. $\mathtt{H}_B \rightarrow_{\mathtt{newChanges,newEI}} \mathtt{H}_A$
   $\mathtt{H}_B$ applies the differences it received in order to establish an up-to-date copy of $\mathtt{H}_A$'s tree $\mathtt{T}'$; it computes the differences between its own latest version and $\mathtt{T}'$, defining a newly "merged" versin of `T` that we call $\mathtt{T}''$; it computes the differences between the newly built tree $\mathtt{T}''$ and $\mathtt{T}'$ in order to inform $\mathtt{H}_A$ of the changes (`newChanges`) that it has to apply to its own copy to build the merged one. It then constructs a new edition identifier `newEI` and finally sends back `newEI` together with `newChanges` to $\mathtt{H}_A$.
6. $\mathtt{H}_A \rightarrow_{\mathtt{ack_A}} \mathtt{H}_B$
7. $\mathtt{H}_B \rightarrow_{\mathtt{ack_B}} \mathtt{H}_A$
   The last two messages are needed just to acknowledge the two hosts that the protocol has been successfully completed. When $\mathtt{H}_B$ receives $\mathtt{ack}_A$, it knows that $\mathtt{H}_A$ possesses the new edition of `T`; it then releases locally the new edition, taking care of adjusting the edition numbers as described in Section 5. The same actions happen on $\mathtt{H}_A$ when

[3] There is always edition 1 at least, as explained in the following section.

```
<complexType name="Order">
 <element name="product" type="string"/>
 <element name="quantity" type="Quantity"/>
</complexType>
<complexType name="Quantity">
 <element name="howmuch" type="decimal"/>
 <element name="resolutor" type="Resolutor"/>
</complexType>
```

Figure 10. Schema definition of the application-specific reconciliation policy.

```
<order>
  <product> Milk</product>
  <quantity>
   <howmuch>1</howmuch>
   <resolutor> add </resolutor>
  </quantity>
</order>
```

Figure 11. XML specification of the application-specific reconciliation policy.

receiving $\texttt{ack}_B$ from $\texttt{H}_B$. The lock on the trees is now removed. If one of them is the owner, it also broadcasts the changes done to the on-line hosts linking to it in order to have a synchronized version.

In case of a failure before step 5, the reconciliation process simply stops: both $\texttt{H}_A$ and $\texttt{H}_B$ re-establish the state they were before the process was started. If the protocol is being stopped between steps 5 and 6, a rollback procedure drops the new edition on both hosts, so actually no reconciliation happens at all. If the protocol fails between steps 6 and 7, $\texttt{H}_A$ rolls-back while $\texttt{H}_B$ completes "successfully", ignoring the fact that actually $\texttt{H}_A$ failed. This is of no harm: next time the two hosts get in reach they will reconcile starting from `lastSharedEI`, because $\texttt{H}_A$ does not possess `newEI`.

Application-Specific Reconciliation

Merging two versions may produce conflicts if both hosts have changed or deleted the same element or attribute. These conflicts need to be reconciled. Unfortunately it is not possible to devise generally applicable conflict resolution strategies that could resolve conflicting updates between replicas without assuming application specific knowledge. XMIDDLE therefore provides the mobile application engineer, who designs the underlying schemas with primitives to specify how confilcts can be resolved in an application specific way.

XMIDDLE supports the definition of confict resolution policies for reconciliation as part of the XML Schema (Fallside, 2000) definition of the data structures that are handled by XMIDDLE itself. This is achieved through the definition of an element type `Resolutor`, as shown in Figure 10. To enable XMIDDLE to resolve the milk bottles conflict on the shopping basket (Section 4), the application designer determines an additional conflict resolution policy in the XML Schema. In particular, the Schema for this example defines type `Resolutor` with values `add`, `last`, `random`, `first`, `greatest`. These policies have an associated priority, defined by the order they appear in the Schema definition.

Figure 11 shows how applications select conflict resolution policies. Referring to our previous example, `add` means that the quantities ordered by the two reconciling members must be

```
<basket>
 <order>
  <product> Milk</product>
   <quantity>
    <howmuch>2</howmuch>
    <resolutor> add </resolutor>
   </quantity>
  </order>
 <order>
  <product> Apple</product>
   <quantity>
    <howmuch>3</howmuch>
    <resolutor> add </resolutor>
   </quantity>
 </order>
</basket>
```

Figure 12. XML file for Member$_C$'s order.

added, therefore three bottles of milk will be included in the reconciled version of the shopping basket. The reconciliation of the shop catalogue among the different PDAs is also performed in a similar way. For the shop catalogue, `resolutor` can be set to `last`, to distribute the latest versions of product catalogue entries.

During the execution of the merge operation, XMIDDLE on host H_B, that is the host that possesses the two trees to reconcile, identifies conflicts by finding changes to attribute or element values. If such a conflict has been detected, the middleware consults the tree and identifies the conflict resolution strategy that has been determined for the attribute or element in question. If the strategy chosen by the two applications is the same, it is simply applied. Otherwise, the policy with the highest priority is chosen, and it is also the one who appears in the merged version on both hosts. If the mobile application designer has not defined a type-specific conflict resolution strategy, XMIDDLE chooses the lastest change, otherwise XMIDDLE determines the attribute or element value by executing the conflict resolution strategy.

We now revisit the collaborative shopping case study of Section 4 in order to see how the reconciliation process actually works. When Member$_A$ and Member$_C$ connect, they consider their linking sets (`LinkedFrom` and `ExportLink`) to identify whether they have common replicas (which they do in our example). The reconciliation method is called by the initiator of the connection (let us say Member$_A$). Reconciliation of the shop catalogue is trivial, as the members would only read, but not modify it: the two members figure out who has the latest version and the other one simply updates his version copying all the changes.

Let us focus now on reconciling the basket, and assume that Member$_A$ has the shopping basket of Figure 11 and that Member$_C$'s basket contains the orders shown in Figure 12. To achieve reconciliation, the ost of Member$_A$ starts by sending the list of all edition identifiers, starting from the current edition until the first one, to Member$_C$ (in this case we suppose Member$_A$ never reconciled after having copied the empty basket from the PC, so he has edition identifier (`1.0`, H_{PC}, `$`) only in addition to the current version he is manipulating). Member$_C$ realizes that edition (`1.0`, H_{PC}, `$`) is the last common one and computes the changes made from that version: she added 2 bottles of milk and 3 apples (as she is also linking to the fruit branch of the catalogue).

```
<diff>
 <graft match="xfl0" type="1" parent="NoRef"
        psib="/*[1]/*[1]">
  <order>
   <product> Apple</product>
    <quantity>
      <howmuch>3</howmuch>
     <resolutor> add </resolutor>
   </quantity>
  </order>
 </graft>
 <replace match="/*[1]/*[1]/*[2]/*[1]/text()[1]"
   type="3">
   <value>2</value>
 </replace>
</diff>
```

Figure 13. Diff of Member$_A$'s and Member$_C$'s baskets.

Member$_C$ sends the update done to the first edition of the basket branch, after calculating them using XMLTreeDiff (Alphaworks, 1998) and locks the tree. The updates are shown in Figure 14 as XMLTreeDiff differences. They are returned in such a way that the merge operation of XMLTreeDiff can take the differences and turn edition (`1.0`, H_{PC}, `$`) into (`2.0`, H_A, H_C) on Member$_A$'s host.

Member$_A$ locks the basket branch and establishes Member$_C$'s update in a new successor version of 1.0. It then uses XMLTreeDiff to compare Member$_A$'s most recent version (the one shown in Figure 11) with the newly established version of Member$_C$'s basket. XMLTreeDiff returns the difference as shown in Figure 13. The merge operation then analyzes these XMLTreeDiff results and identifies that there are two differences. The first one `graft` is a new order that is to be inserted. This can be merged into Member$_A$'s basket by XMLTreeDif without causing a conflict. The second difference is a `replace`, which indicates a conflict. The conflicting node is the `howmuch` element identified by the XPath expressin of the `match` attribute. Instead of applying the replace operation as it is, the merge operation consults the application-specific conflict resolution strategy in the document and as a resolution changes the `value` element of the `replace` node to `3` (to cater for the additional bottle of milk that was in the `howmuch` element of Member$_A$'s basket).

The merge operation then applies the differences to Member$_A$'s shopping list by calling XMLTreeDiff's merge operation. Finally, it computes the differences between the result and Member$_C$'s list to be sent back to Member$_C$ together with the new common version number. In this way we have fully reconciled the two versions on the PDAs.

6.2. Linking Protocol

This protocol is a simplification of the previously described one. In fact, we can think of the `link` operation as a reconciliation between a tree `T` and an empty one T_0 considered as edition 0 of `T`. The output of the reconciliation process causes no changes to the linked tree `T`, while the empty tree T_0 becomes a full copy of `T`, with no conflicts to reconcile at all.

1. $H_B \rightarrow_{\texttt{LinkedFrom}_B,\texttt{ExportLink}_B} H_A$

```
<?xml version="1.0"?>
<diff>
  <add match="xfl0"  type="1" name="order" parent="/*[1]" psib="xfl1"/>
  <add match="xfl3"  type="1" name="quantity" parent="xfl0" psib="xfl2"/>
  <add match="xfl5"  type="1" name="resolutor" parent="xfl3" psib="xfl4"/>
  <add match="xfl6"  type="3" parent="xfl5"> <value> add </value> </add>
  <add match="xfl4"  type="1" name="howmuch" parent="xfl3"/>
  <add match="xfl7"  type="3" parent="xfl4"> <value>3</value> </add>
  <add match="xfl2"  type="1" name="product" parent="xfl0"/>
  <add match="xfl8"  type="3" parent="xfl2"> <value> Apple</value> </add>
  <add match="xfl1"  type="1" name="order" parent="/*[1]"/>
  <add match="xfl10" type="1" name="quantity" parent="xfl1" psib="xfl9"/>
  <add match="xfl12" type="1" name="resolutor" parent="xfl10" psib="xfl11"/>
  <add match="xfl13" type="3" parent="xfl12"> <value> add </value> </add>
  <add match="xfl11" type="1" name="howmuch" parent="xfl10"/>
  <add match="xfl14" type="3" parent="xfl11"> <value>2</value> </add>
  <add match="xfl9"  type="1" name="product" parent="xfl1"/>
  <add match="xfl15" type="3" parent="xfl9"> <value> Milk</value>
  </add>
</diff>
```

Figure 14. Result of TreeDiff between edition 1.0 (i.e., the empty basket) and the current shopping list of Member_C.

2. $\mathtt{H}_A \rightarrow_{\mathtt{LinkedFrom_A,ExportLink_A}} \mathtt{H}_B$
   The first two steps are exactly the same as in the reconciliation protocol: the two hosts exchange their `LinkedFrom` and `ExportLink` sets in order to find out whether they share information. We assume here that they do not share anything, in order to illustrate this new case. For example, $\mathtt{H}_B$ may decide to link a subtree `T` belonging to $\mathtt{H}_A$ after having seen it listed in $\mathtt{ExportLink}_A$, or we may think that $\mathtt{H}_B$ already linked to `T` while disconnected from $\mathtt{H}_A$.
3. $\mathtt{H}_B \rightarrow_{\mathtt{T,(0,\$,\$)}} \mathtt{H}_A$
   This message is exactly the same as in the reconciliation protocol, but the list of edition identifiers contains only one entry, (0, $, $).[4]
4. $\mathtt{H}_A \rightarrow_{\mathtt{(1,H_A,\$),LastEdition,activeChanges}} \mathtt{H}_B$
   When $\mathtt{H}_A$ receives the tuple (0, $, $), it knows that $\mathtt{H}_B$ wishes to link `T`, and replies with the latest edition of the tree together with a list of changes previously broadcasted but not already released in an edition. $\mathtt{H}_B$ can now store this latest edition and apply the changes in order to synchronize with $\mathtt{H}_A$. Since now on $\mathtt{H}_B$ receives all the changes to `T` broadcasted by $\mathtt{H}_A$. It is worthwhile noticing that $\mathtt{H}_A$ sends also the very first edition of `T` to $\mathtt{H}_B$; doing so, $\mathtt{H}_B$ will be able to reconcile with any other hosts linking to the same tree, as there is always at least one common edition, the first one.

6.3. Disconnection Protocol

The disconnection protocol involves only the host who is disconnecting, for instance $\mathtt{H}_B$, so we would actually call it "disconnection procedure" rather than "disconnection protocol".

[4] We use the tuple (0, $, $) to identify the empty edition.

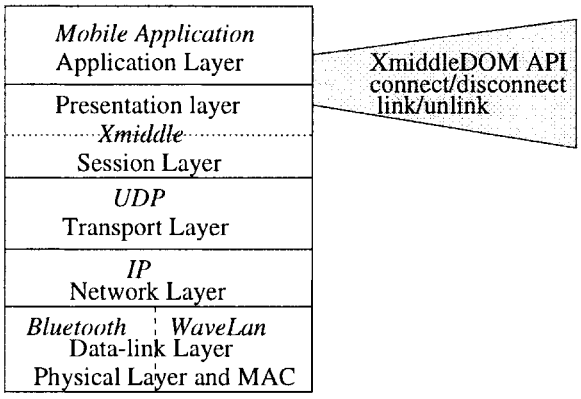

Figure 15. The protocol stack for mobile environments using XMIDDLE.

This protocol is initiated by the application in case of an explicit disconnection, while it is started by XMIDDLE in case of an implicit disconnection. In both cases, for each version not yet released, the host releases it: the versioning process is started and finally the tree is stored. All the edition identifiers issued in this procedure will have the form (`editionNumber`, H_B, \$). There is no need for H_B to broadcast a message to notify of its imminent disconnection: the middleware of the hosts connected (i.e., listed in the `InReach` set) will take care of the fault, initiating a disconnection procedure that releases all the versions of branches linked to the disconnected host.

It is worth noticing that this protocol completely disconnects a host from the network when it is invoked by the application; on the contrary, it may disconnect a host H_A from another host H_B while leaving H_A still connected to H_C and H_D, when invoked by the middleware of H_A as a consequence, for instance, of a movement.

7. The XMIDDLE Architecture

We now present an overview of the XMIDDLE architecture, which follows the ISO/OSI reference model. Figure 15 shows the protocol stack. As shown, XMIDDLE implements the session and presentation layers on top of standard network protocols, such as UDP or TCP, that are provided in mobile networks on top of, for instance, a Bluetooth data-link layer (i.e., Logical Link Control and Adaptation Protocol) and MAC and physical layer (i.e., Bluetooth core which is based on radio communication). Our current prototype is however based on UDP upon Wireless Lan (WaveLan (Technologies, 2000)), which is another possible option.

The presentation layer implementation maps XML documents to DOM trees and provides the mobile application layer with the primitives to link, unlink and manipulate its own DOM tree, as well as replicas of remote trees. The session layer implementation manages connection and disconnection.

Figure 16 refines the presentation and session layer implementations of XMIDDLE. The *Xmiddle Controller* is a concurrent thread that communicates with the underlying network protocol and handles new connections and disconnections, triggers the reconciliation procedures and handles reconciliation conflicts according to application specific policies. As XMIDDLE is entirely implemented in Java, it relies on a Java Virtual Machine (JVM). A large variety of JVMs have been implemented for mobile devices. The Symbian operating system for the third

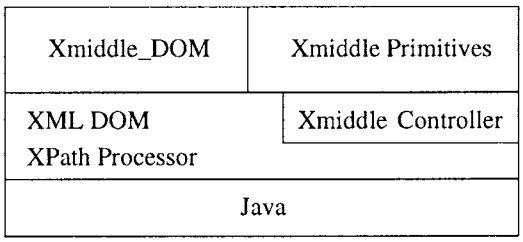

Figure 16. The XMIDDLE architecture.

generation of mobile phones, for examples, has a Java Virtual Machine built in. Likewise, Sun provides a minimal kernal virtual machine (KVM) implementatin for Palm PDAs.

The *Xmiddle Primitives* API provides mobile applications with operations implementing the XMIDDLE primitives, such as link, unlink, connect and disconnect. The ability to link to trees from other devices introduces a client/server dependency between mobile hosts. We refer to the host which a tree is linked from as the *server host* and the host that links the three as a *client host*. The XMIDDLE implementation maintains this client/server relationship in the `LinkedFrom` and `LinkedBy` tables that are kept on each host (they correspond to the sets with the same names defined in Section 2). The `LinkedFrom` table also needs to keep track of the host that owns a subtree in order to allow the application to be able to request updates from that host; this is done using XPath. It is also necessary to the hosts that have linked to a tree for being able to broadcast updates when the hosts are in reach.

The *Xmiddle_DOM* component provides the XMIDDLE implementation of the DOM to mobile applications. We now proceed with a detailed description of the XMIDDLE primitives.

7.1. XMIDDLE PRIMITIVES

Connect

Each entry in the `LinkedFrom` and `LinkedBy` tables identifies a remote host as well as a specific branch of that host's XML tree. The `ExportLink` table identifies the branches of the local tree that can be linked to from remote hosts. The `InReach` table contains the list of hosts in reach. The `connect` primitive allows an application to notify the hosts in reach that it is re-connected. The notified hosts will then update their `InReach` tables. Upon reconnection the host starts the reconciliation protocol with all the hosts in reach which are linking/linked to some parts of its tree. After reconciliation, and provided that the connection is still available, the host maintains the *on-line mode update* status: it broadcasts all changes to its tree to other hosts included in the `LinkedBy` table and the client hosts send requests for changes to the server.

Disconnect

The `disconnect` primitive allows a host $\mathtt{H}_A$ to explicitly decide to work off-line. Apart from the explicit disconnection of $\mathtt{H}_A$, the unreachability of a host $\mathtt{H}_B$ from $\mathtt{H}_A$ can be obtained implicitly when one of the two hosts moves away. The disconnection process changes the content of the `InReach` table and the disconnection protocol is invoked in order to handle the tree changes and information caching.

Link
Linking a tree from a remote host is achieved by calling the XMIDDLE operation `link`. Its arguments indicate the server host and the complete path to the branch. Furthermore, they identify the local "mounting" point. During execution of the `link` operation XMIDDLE records the linking details in the `LinkedFrom` table. Note that the link primitive can be used independently from the connection status in order to indicate an intention to share some information with another host. When linked and connected to a remote client host, the server host records the name of the client host, the branch it is linking to, and the linking point in the `LinkedBy` table. This is used for broadcasting changes from the server host during connection with the client host.

Unlink
The unlink primitive modifies the local `LinkedFrom` table "unmounting" a particular branch of a tree (maybe because the application does not need it anymore).

DOM Operations
XMIDDLE provides all the operations specified for tree traversal and manipulation in the DOM Level1 Recommendation. All operations access and manipulate the local XML tree.

For access (i.e., read) operations, such as `firstChild, parentNode`, and `nextSibling`, that return data from either the owned tree of the host, or any linked tree, the XMIDDLE DOM interface just accesses the local DOM tree (or replica) using the Apache DOM implementation. No remote communication is needed to perform these generally frequently used access operations.

For update operations of the tree, we have to distinguish several cases. If a host wants to update its own tree, the update is performed by calling the underlying Apache DOM implementation and then broadcasting the changes to all the hosts connected and that are linked to the changed branch (i.e., the `LinkedBy` table is interrogated). If an application wishes to update a remote branch that is linked from another host, we again have to distinguish two cases. If the owner is not within reach we perform the changes on that version locally using the Apache DOM implementation. The reconciliation protocol upon reconnection will synchronize the versions of the common branches. If the owner host is within reach, we request it to perform the update and wait until we receive the notification of the changes before performing them on its replica. The update requests that the server host receives are queued together with the ones issued by itself, and then processed with a FIFO policy. If a reconciliation protocol is started by a re-connecting host, this has priority, the request queue is flushed and after reconciliation the resulting changes are broadcasted to the hosts in reach linking to the branch.

7.2. IMPLEMENTATION

We have been implementing the XMIDDLE platform and an addressbook application utilising the middleware, on a Compaq iPAQ PDA running Linux. Figure 18 shows the use interface of the addressbook application. We used Xerces as the DOM implementation and Xalan as the XPath expression processor, as provided by the Apache Software Foundation. We found that these packages are not suitable for running under the sun CDLC virtual machine of Java 2 Micro Edition, so we resolved to using Java 2 Standard Edition version 1.3.1. Figure 17 shows the versions of all the main software components used. Note that the XMIDDLE platform requires just 156 KB of space, while the addressbook application requires 24 KB.

Package	Version	Storage	Notes/Description
Linux	Kernel 2.4.3	-	Distribution: Familiar 0.4 provided by http://www.handhelds.org
Java Runtime Environment	1.3.1-rc1 ARM	18MB	Beta version of the Java 2 runtime environment, provided by http://www.blackdown.org. Note that the size can be reduced, as this number includes all the classes in the Java classpath. Also note that this version of java does not unclude a Just in Time (JIT) compiler for the ARM processor that the iPAQ uses.
Xerces	1.4.1	1.8MB	Provided by the Apache Software Foundation (http://xml.apache.org)
Xalan	2.2.D6	416 KB	Provided by the Apache Software Foundation (http://xml.apache.org)
XMIDDLE Framework	1.0	24KB	Provides an abstract framework of the xmiddle platform
XMIDDLE Implementation	0.7	132KB	The core xmiddle implementation.
AddressBook	0.3	24KB	The address book application built utilising xmiddle

Figure 17. A table showing the packages, versions and storage requirements of the various components used.

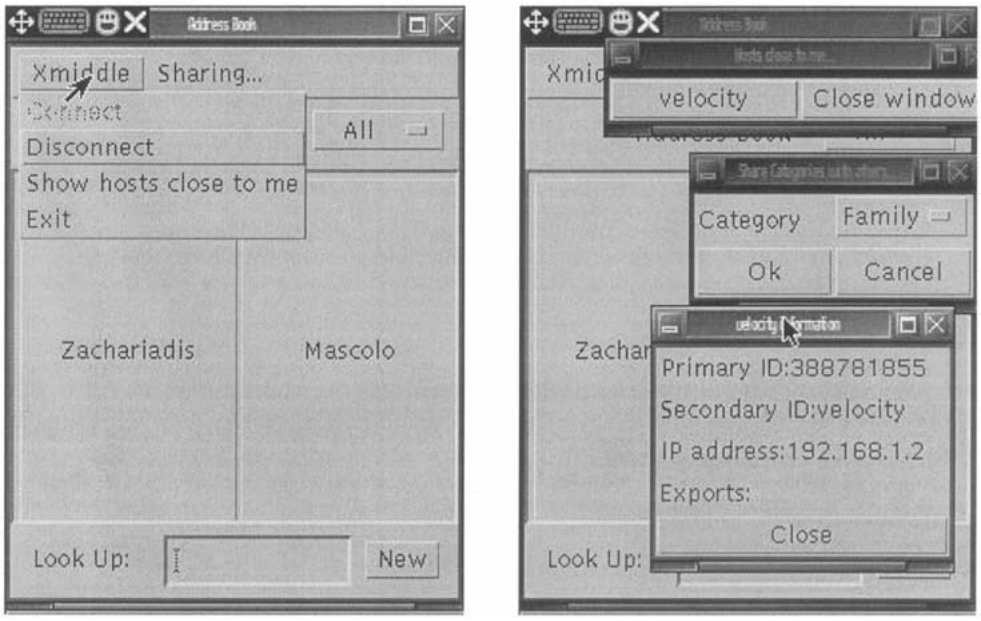

Figure 18. (a) The addressbook application running on the IPAQ. (b) Showing more features of the application, including sharing data and discovering host information.

With the sample document used, we found the addressbook along with the middleware to require approximately 14 MB of RAM and startup time for the middleware services and the application is approximately 22 seconds. Note that in all measurements, XML validation was turned off in the parser. Moreover, the virtual machine used does not have a Just In Time (JIT) compiler for the ARM processor that the iPAQ uses.

Further tests conducted concluded that the major bottleneck in these results was the Xerces parser, which we found not to be suitable for mobile devices. Xerces 2, not available at the time of writing, is supposed to be much faster and more efficient than the current versions

available, and thus we plan to switch our implementation to that, when available. We also plan to investigate the possibility of developing an XPath processor and a DOM implementation specifically targeted to mobile devices. We feel that this is an area which has yet to be targeted, and that we will be able to have our prototype running using Java 2 Micro Edition.

8. Discussion and Related Work

We have described XMIDDLE and shown its architecture. Through a case study we have illustrated how it is used and shown the details of the tree reconciliation algorithms and linking used for data synchronization among the mobile devices. Synchronization and data locking have been described as main problems in wireless environments by Imielinski and Badrinath in (Imielinski and Badrinath, 1994). XMIDDLE offers a possible solution.

We focus our interest on ad-hoc networks where host configurations are relative and dynamic. No discovery services are set-up as in Jini (Arnold et al., 1999) as all the hosts have the same capabilities. They are able to reconfigure their own connection groups while they move, through connection and disconnection with the other hosts.

Tuple space based systems for logical and physical mobility such as JavaSpaces (Freeman et al., 1999), Lime (Murphy et al., 2001), Limbo (Davies et al., 1997), T Spaces (IBM), and Mars (Cabri et al., 1998) exploit the decoupling in time and space of these data structures in the mobility context where connect and disconnect are very relevant and frequent operations. However, tuple spaces are very general and loose data structures, which do not allow complex data organizations and therefore do not fit all the application domains. XML allows us to introduce hierarchy of data and to address specific paths in the structure so that more elaborated operations can be performed by the applications. The value of XML in structuring data has already been recognized and some work has also been carried out to integrate tuple spaces and XML: in Cabri et al. (2000), a mobile agent system based on tuple spaces is integrated with XML for the encoding of data. This allows a more structured way of dealing with data communication, while introducing flexibility in the data treatment. In that paper, however, XML is only used for data formatting. Tuples are translated into XML files and stored into a data-space. In Abraham et al. (1999), XML is used to create a lightweight repository of XML documents, based on IBM's T Spaces. This repository supports XML (DOM) oriented queries. XML documents are somehow stored as tuples in the tuple spaces. TSpaces recently offered direct support for storage and indexing of XML documents. This is done by transforming XML documents into a tree of TSpaces tuples, linked internally via pointers.

An additional disadvantage of tuple-spaced based systems is in term of synchronization capabilities. Tuple-spaces are multi-sets, which means every tuple can be duplicated in the space. Whenever two or more devices, which replicate a piece of data (represented as a tuple), disconnect and modify it the reconciliation process of rejoining the tuple spaces during reconnection becomes an unnatural operation (due to the multi-set property of tuple spaces).

The issue of data replication and synchronization has been addressed in the context of distributed file systems by Coda (Satyanarayanan et al., 1990), which adopts an application-transparent adaptation technique, and its successor Odyssey (Satyanarayanan, 1996), which enables application-aware adaptation. Compared to these approaches, XMIDDLE firstly defines a different level of granularity of the data that can be moved across mobile devices, that is, parts of an XML document, as small as we wish, as opposed to whole files. This may have a relevant impact when dealing with slow and/or expensive connection. Moreover, we do not

assume the existence of any server that is more capable and trustworthy than mobile clients, as we target pure ad-hoc network configurations. Finally, the use of XML adds semantic to the replicated data, against the uninterpreted byte streams of files; this added semantics can then be exploited to provide better confict detection and resolution policies from an application point of view (as shown in Section 6).

XMIDDLE uses only XML trees as data structures and exploits the power of the nature of the data structure with specific operations; for instance, the linking primitive facilitates off-line sharing of information, which is very valuable in mobile computing contexts where hosts have the need to move away from the source of information even if they may want to continue to work on the downloaded data. Reconciliation mechanisms are needed to maintain a certain level of consistency and to support synchronization. Existing mobile computing middleware systems do not address this issue and a consortium (i.e., SyncML (SyncML, 2000)) has been established in order to provide standards for synchronizing data in mobile computing. SyncML provides a set of specifications for the standardization of synchronization of data (in any format) between different devices, using WSP, HTTP, or Bluetooth protocols, XMIDDLE uses tree structures for representing data and defines protocols that take advantage of this format. SyncML focuses on peer-to-peer synchronization, where a client/server relationship is always established among the devices. No ad-hoc networking setting is supported by SyncML, whereas XMIDDLE also supports reconciliation of different clients that possess replicas of specific branches of an XML tree. SyncML also defines reconciliation policies for data synchronization. However, the policies are either on the server or client side. The case in which the client wants to indicate how to reconcile data to the server is not supported. As we have shown in the case study analysis, hosts sometimes need to specify different reconciliation policies and some priority structure among the policies is needed to actually choose which policy to apply. Unlike SyncML, XMIDDLE avoids the need for application to log every change they apply to shared data. Instead XMIDDLE uses a versioning system to make this aspect transparent. SyncML, on the contrary, leaves the logging to the application level. Security and authentication aspects are investigated in the SyncML specification which XMIDDLE does not tackle yet. However, some authentication mechanisms similar to the one of SyncML could be put in place in XMIDDLE, too.

The aim of Globe (v. Steen et al., 1999) is to provide an object based middleware that scales to a billion users. To achieve this aim, Globe makes extensive use of replication. Unlike other replication mechanisms, such as Isis (Birman, 1997), Globe does not assume the existence of an application independent replication strategy. It rather suggests that replication policies have to be object-type specific, and therefore they have to be determined by server object designers. In Globe each type of object has its own strategy that pro-actively replicates objects. XMIDDLE policies definition follows this approach.

The XMIDDLE strategy for data synchronization exploits well established techniques and tools for replication and reconciliation on trees (Tai, 1979; Alphaworks, 1998). In Shapiro et al. (2000), some formal work on application-independent reconciliation has been carried out, which also focuses on a structured way for applications to influence data reconciliation choices, XMIDDLE exploits semantic knowledge about elemental types; a set of reconciliation primitives is defined in XMIDDLE, as described in Section 6, and the mobile application engineer can specify the way these primitives are combined to determine an application-specific reconciliation policy. In this way we can ease the burden of applications, relying as much as possible on the middleware, while, at the same time, providing for the application semantics and user policies. This differentiates XMIDDLE from systems like CVS (Cederqvist

et al., 1992) and Bayou (Petersen et al., 1997). CVS is a source code versioning tool that leaves everything in the hands of the user; conflicts are detected based on updates done in the same line of the file by different users, and the conflict resolution is left to the user. Bayou reconciles application-specific information in an application-independent way, preventing the application from influencing the outcome of the reconciliation process. Bayou's philosophy is the traditional middleware one, which call for complete *transparency*.

The XMIDDLE reconciliation algorithm is relying on versioning mechanisms. Like text-based versioning systems, such as RCS (Tichy, 1985), we store and transmit differences as shown in Figures 13 and 14 to minimize the transmission load during reconciliation: only the updates from the last common version are exchanged between the hosts. Unlike text-based versioning, however, the differences the XMIDDLE implementation is able to obtain from XMLTreeDiff are more precise and semantically richer. This is because the differencing algorithms are able to take attribute and element, as well as their arrangements in trees into account. This generally leads to a smaller number of conficts than in text-based differencing tools.

9. Further Work and Concluding Remarks

The growth of the recent mobile computing devices and networking strategies call for the investigation of new middleware that deal with mobile computing properties such as disconnection, low/expensive bandwidth, scarce resources and in particular battery power, in a natural way. XMIDDLE is one possible answer to these needs that focuses on data replication and synchronization problems and solves them exploiting reconciliation strategies and technologies.

The implementation of the current prototype of XMIDDLE is based on Wireless LAN and UDP, however we plan to migrate the system to Bluetooth for more testing. Every host has a unique ID, which is used for enumerating the tree versions in a consistent way (as described in Section 5). In an earlier versin of the prototype we used the XMLTreeDiff tool developed by IBM (Alphaworks, 1998) but later we decided to implement our own one that does not "optimize" the results as the IBM version does as this was not needed in XMIDDLE. The linking of a tree is currently implemented replicating the linked branch locally. However, we plan to use different linking policies, depending on the available bandwidth. That is, avoiding caching of the linked tree when the two hosts are in reach and good bandwidth conditions are matched. A weakness of the current reconciliation protocol implementation is that it does not cover the case when two hosts that link to the same branch get in reach. The prototype currently reconciles the two replica, but if further modifications are done by either of the hosts, these changes are not broadcasted to the other host but instead the replicas become inconsistent again until either an implicit or an explicit reconciliation is started. We chose to implement the reconciliation this way in order to avoid a heavy leader election protocol implementation but realize that we might have to revisit this decision after we have gained some practical experience with it.

The reconciliation and linking policies can be refined, especially considering the case where trees become graphs through XPath expressions that create links (pointers) inside the tree. We are also considering more case studies to deal with the conflict resolutions in a mixed application/non-application oriented fashion. The definition of policies for inconsistency resolution during the reconciliation process may also be considered as quality of

service specification. By defining the level of consistency the application needs on specific data it is possible to specify different "qualities of reconciliation". We itnend to investigate this approach further.

Security policies can also be established in order to limit the access of hosts on XML trees. For instance, specific branches of the trees may be defined as accessible to all the hosts while other branches may be accessible only to particular hosts. This can be done enriching the syntax of the `LinkExport` table which allows a host to make only some subtrees remotely accessible while retaining exclusive access to other subtrees. Digital signatures and common security strategies (i.e., passwords and public/private keys) could be applied as well in order to guarantee further levels of security. Issues of fault tolerance which we tackle only partially are in our agenda as well.

The use of XML and XPath for data formatting has advantages not only at the level of the tree structure and at the use of readily available technologies, but also at the information rendering level as XSL and WAP could be integrated in order to customize the display of the data for different mobile devices.

In Mascolo et al. (2001), we used XML for the implementation of a fine-grained code mobility approach which allows single lines of code to be transferred among hosts in an incremental manner. XMIDDLE allows data sharing through XML; however, using the approach presented in the mentioned paper we could provide code sharing and mobility using the same XML format. This feature would power XMIDDLE with more flexibility and extensibility: we plan to look into this aspect.

Tuple spaces based systems allow notification of events on the tuple spaces in different ways (e.g., transactions and reactions). We plan to extend XMIDDLE by introducing some event notification mechanisms that allow hosts to register for events on trees. At the moment a basic event notification mechanism is in place for connected hosts to be notified about the modification of linked tree branches, but some extensions can be developed.

In conclusion, XMIDDLE is an example of a reflective middleware (Eliassen et al., 1999). XMIDDLE abandons replication transparency as we believe that in the challenging mobile computing environments middleware systems have to take advantage of application-specific information to achieve an acceptable performance, usability and scalability. We consider our effort on XMIDDLE to be just the first step in that direction and believe that a number of other forms of transprency have to be given up, too. Location transparency, for example, may have to be discontinued to provide location aware services. In general, this will lead to a new class of context-aware applications (Capa, Emmerich and Mascolo, 2001), which can influence the way middleware implements interactions between mobile components based on the context in which the components operate.

Moreover, mobile ad-hoc network research is recently investigating behaviour and routing in a multi-hop scenario, where hosts act as router allowing transitive communication. We think XMIDDLE can be expanded to deal with these protocol and we plan to investigate this issue further.

Acknowledgements

We would like to thank Jon Crowcroft, Adam Greenhalgh, Steve Hailes, Gruia-Catalin Roman, and Vassilis Rizopoulos for the helpful discussions on the topic and their comments on

a draft of this paper. We also thank Christian Nentwich for the non-optimizing XMLTreeDiff code.

References

J. Abraham, H. Le and C. Cedro, "XML Repository in T Spaces and UIA Event Notification Application", http://www.cse.edu/projects/1998–99/project19, 1999.

I. Alphaworks, "SML TreeDiff", http://www.alphaworks.ibm.com/tech/xmltreediff, 1998.

V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A.L. 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, 1998.

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

K.P. Birman, *Building Secure and Reliable Network Applications*, Manning Publishing, 1997.

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, 1998.

G. Cabri, L. Leonardi and F. Zambonelli, "Reactive Tuple Spaces for Mobile Agent Coordination", in *Proceedings of the 2nd International Workshop on Mobile Agents (MA 98)*, Springer, 1998.

G. Cabri, L. Leonardi and F. Zambonelli, "SML Dataspaces for Mobile Agent Coordination", in *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000)*, Como, Italy, ACM Press, 2000.

L. Capra, W. Emmerich and C. Mascolo, "Middleware for Mobile Computing: Awareness vs. Transparency (position paper)", in *Int. 8th Workshop on Hot Topics in Operating Systems*, 2001.

J. Clark and S. DeRose, "XML Path Language (XPath)", Technical Report, http://www.w3.org/TR/xpath, World Wide Web Consortium, 1999.

N. Davies, S.P. Wade, A. Friday and G.S. Blair, "Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications", in *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pp. 291–302, 1997.

F. Eliassen, A. Andersen, G.S. Blair, F. Costa, G. Coulson, V. Goebel, O. Hansen, T. Kristensen, T. Plagemann, H.O. Rafaelsen, K.B. Soikoski and W. Yu, "Next Generation Middleware: Requirements, Architecture and Prototypes", in *Proceedings of the 7th IEEE Workshop on Future Trends in Distributed Computing Systems*, IEEE Computer Society Press, pp. 60–65, 1999.

W. Emmerich, *Engineering Distributed Objects*, John Wiley & Sons, 2000.

D.C. Fallside, "XML Schema", Technical Report, http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium, 2000.

E. Freeman, S. Hupfer and K. Arnold, *JavaSpaces[tm] Principles, Patterns, and Practice*, Addison-Wesley, 1999. D. Gelernter, "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80–112, 1985.

IBM, "T Spaces", http://almaden.ibm.com/cs/TSpaces.

T. Imielinski and B.R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management", *Communications of the ACM*, Vol. 37, No. 10, pp. 18–28, 1994.

L. Capra, W. Emmerich and C. Mascolo, "Reflective Middleware Solutions for Context-Aware Application", in *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 01)*, 2001, to appear.

C. Mascolo, L. Zanolin and W. Emmerich, "XMILE: an XML Based Approach for Incremental Code Mobility and Update", *Automated Software Engineering*, 2001, to appear.

R. Mettala, "Bluetooth Protocol Architecture", http://www.bluetooth.com/developer/whitepaper/, 1999.

A.L. Murphy, G.P. Picco and G.-C. Roman, "LIME: A Middleware for Physical and Logical Mobility", in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, 2001.

P. Cederqvist et al., "Version Management with CVS", 1992.

K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication", in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, ACM Press, pp. 288–301, 1997.

M. Satyanarayanan, "Mobile Information Access", *IEEE Personal Communications*, Vol. 3, No. 1, 1996.

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, 1990.

M. Shapiro, A. Rowstron and A. Kermarrec, "Application-Independent Reconciliation for Nomadic Applications", in *Proceedings of European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding: Denmmark, SIGOPS, 2000.

SyncML, "Building an Industry-Wide Mobile Data Synchronization Protocol", http://www.syncml.org/technical.htm, 2000.

K. Tai, "The Tree-to-Tree Correction Problem", *Journal of the ACM*, Vol. 29, No. 3, 422–433.

L. Technologies, "WaveLan", http://www.wavelan.com, 2000.

W.F. Tichy, "RCS – A System for Version Control", *Software – Practice and Experience*, Vol. 15, No. 7, pp. 637–654, 1985. M. v. Steen, P. Homburg and A.S. Tanenbaum, "Globe: A Wide-Area Distributed System", *IEEE Concurrency*, pp. 70–78, 1999.

Cecilia Mascolo (http://www.cs.ucl.ac.uk/staff/c.mascolo) holds a Laurea degree in Science dell' Informazione and a Ph.D. in informatica from the University of Bologna, Italy. In 1999, she spent a year as a visiting academic at the Department of Computer Science at Washington University, Saint Louis. In February 2000, she became a research fellow at the Department of Computer Science at University College London and joined the academic staff of the Department as a lecturer in computer science in February 2001. She has published extensively in the areas of software engineering, mobile computing, mobile code, ad-hoc and active and peer to peer networks. Cecilia is also interested in the use of mark-up languages for mobile computing applications. She is principal investigator and co-investigator in three projects related to mobile computing middleware and middleware for active networks.

Licia Capra (http://www.cs.ucl.ac.uk/staff/l.capra) holds a Laurea degree in informatica from the University of Bologna, Italy. From May 2000 to September 2000, she worked as a research assistant in the Department of Computer Science at University College London and since September 2000 she is a Ph.D. student in the same department. Her research focuses on the design and prototyping of reflective middleware for mobile computing. Licia is also working as a software engineer for the Zuhlke Technology Group (www.zuhlke.com).

Stefanos Zachariadis (http://www.cs.ucl.ac.uk/staff/s.zachariadis) received his B.Sc. in computer science from University College London, United Kingdom, in 2001. He is currently a Ph.D. student at the same institution, researching the use of logical mobility in physically mobile environments. His interests include ad-hoc networking, mobile computing middleware, mobile code techniques and peer to peer communications.

Wolfgang Emmerich (http://www.cs.ucl.ac.uk/staff/w.emmerich) received his M.Sc. from University of Dortmund, Germany in 1990 and his Ph.D. from University of Paderborn in 1995. His Ph.D. thesis was on database support for integrated software engineering environments. Wolfgang was a research assistant in the Department of Computer Science at Dortmund between 1990 and 1995. After his Ph.D. he joined City University, London as a lecturer and developed an interest in software engineering for distributed object-based systems. In November 1997, he joined UCL where he now has the position of a senior lecturer. His reserch interests are in design of distributed objects and middleware for mobile systems. Wolfgang was a senior consultant at the OMG Representative for Central Europe, where he developed his distributed object consulting expertise. He has become a recognised expert in the area of software engineering for distributed objects and is the author of a text book on "Engineering Distributed Objects" published by John Wiley & Sons. Wolfgang has extensively consulted in the European software engineering industry and is now a partner of the Zuhlke Technology Group and a senior consultant and director of Zuhlke Engineering (U.K.) Ltd. in London.