Mobility Support for Replicated Real-Time Applications

Jörg Roth

University of Hagen
Department for Computer Science
58084 Hagen, Germany
Joerg.Roth@Fernuni-hagen.de

Abstract. Replicated real-time applications such as co-operative document editors have to continuously update a shared state, thus require low network delays. If we use such applications in mobile and weakly connected environments, state information often cannot be broadcasted immediately, and thus it is difficult to maintain consistency. We discuss this problem with the help of *DreamTeam*, our framework for distributed applications, which we extend to the mobile version *Pocket DreamTeam*. The DreamTeam environment allows the developer to generate replicated applications (e.g., collaborative diagram tools, multi-user text editors, shared web browsers) in the same way as single user applications, without struggling with network details or replication algorithms. For our mobile extension, we suggest an architectural decomposition according to the *remote proxy pattern*. This architecture has a number of benefits: it tolerates weakly connected devices and allows a developer to heavily re-use existing stationary applications.

1 Introduction

Replicated real-time applications play a major role for e.g. shared document editors, co-operative software development environments or shared workspaces. Replicated states are a basis for collaborative multi-user applications, which allow geographically distributed teams to collaborate without significant time delays. Replicated real-time applications store their state information on each participating site without the need for a central server. They have, compared to applications with centralised architectures, a lower response time, since state information is available locally. The network and processing load is distributed among all sites, thus performance bottlenecks are avoided.

To support developers of replicated real-time applications or synchronous groupware we developed the *DreamTeam* platform [6, 10]. We successfully used DreamTeam for practical software courses and diploma theses at the University of Hagen. There exists a huge variety of about 20 DreamTeam applications such as a distributed sketch tool, a diagram tool, text editor, a collaborative slide presentation program, a brainstorming tool and a group web browser.

Currently, there exists a growing market for mobile devices such as PDAs, mobile phones, electronic pens etc. Upcoming communication technologies like UMTS and

Bluetooth promise new services for mobile communication. We strongly believe that mobile computing combined with distributed applications provides a great potential, thus we want to extend the DreamTeam architecture to support mobile users. The corresponding DreamTeam extension called *Pocket DreamTeam* should support weakly connected devices with reduced computational power and limited input/output capabilities. The Pocket version and the stationary version of DreamTeam should run inside the same network. To save development costs, existing shared applications based on the old DreamTeam platform should run without any modification.

The problems to create a mobile platform extension are manifold. In this paper we focus on two major problem areas:

- Replication strategies based on low network delays and high network reliability are not suitable for wireless networks, which still suffer from temporary disconnections and high latency. Thus, we adapted the existing DreamTeam replication mechanism [10] to support mobile devices.
- A developer of mobile replicated applications has to develop for stationary as well as for mobile computers with completely different characteristics. To reduce development costs, we introduce an approach, which allows the developer to re-use code of the functional core for both platforms.

Before we present our approach in more detail, we discuss related work.

2 Related Work

Several toolkits have been developed so far to address consistency problems in mobile environments. *Coda* [4] provides a distributed file system similar to NFS, but allows disconnected operations. Applications based on Coda are fully mobility transparent, i.e. run inside a mobile environment without any modification. Disconnected mobile nodes have access to remote files via a cache. Operations on files are logged and automatically applied to the server when the client re-connects. Coda applications can either define themselves mechanisms for detecting and resolving conflicts or ask the user in case of conflicts. A follow-on platform, *Odyssey* [9], extends data distribution to multimedia data such as video or audio data. To support real-time data, bandwidths and available resources have to be monitored. Odyssey applications are mobility aware.

Rover [3] supports mobility transparent as well as mobility aware applications. To run without modification, network-based applications such as web browsers and news readers can use network proxies. The development of mobility aware applications is supported by two mechanisms: *relocated dynamic objects* (*RDOs*) and *queued remote procedure calls* (*QRPC*). RDOs contain mobile code and data and can reside on a server as well as on a mobile node. During disconnection, QRPCs are applied to cached RDOs. As in Coda, operations are logged and applied to server data after reconnecting.

Bayou [19] provides data distribution with the help of a number of servers, thus segmented networks can be handled. In contrast to Coda, replicated records are still accessible, even when conflicts were detected but not resolved. Bayou applications have to provide a conflict detection and resolution mechanism, thus user intervention is not necessary. Bayou is not designed to support real-time applications.

Sync [7] allows asynchronous collaboration between mobile users. Sync provides collaboration based on shared objects, which can be derived from a Java library. As in Bayou, data conflicts are handled by the application. Sync applications have to provide a *merge matrix*, which contains a resulting operation for each pair of possible conflicting operations. With the help of the merge matrix, conflicts can be resolved automatically.

Lotus Notes [5] has not primarily been designed for mobile computers, but allows replicated data management in heterogeneous networks. Nodes can be disconnected and merge their data after re-connection. Data in Lotus Notes have a record structure. Fields may contain arbitrary data, which are transparent to Notes. Records can be read or changed on different nodes simultaneously. When re-connecting, users resolve conflicting updates.

A completely different approach to support mobile users introduces *Pebbles* [8]. It allows users to remotely control applications running on a server. It follows a collaboration and mobility transparent concept. Instead of using the mouse and keyboard directly, input is taken from the touch screen and handwriting area. It is possible to remotely control off-the-shelve applications (e.g. MS Word) with handheld devices.

As a last example, we want to mention our own mobile platform *QuickStep* [11, 14]. The replication mechanism based on the *database* abstraction integrated into most handheld operating systems [1]. The consistency strategy relies on a strong connection between data rows and involved users. Although QuickStep was primarily designed to exchange well-structured record-oriented data among a group of mobile handheld users, it highly influenced our second platform Pocket DreamTeam.

3 Pocket DreamTeam

The original DreamTeam environment mainly consists of a huge hierarchical class library with approx. 200 classes and 125000 lines of code. The stationary part is entirely written in Java, which can be run on many operating systems, e.g. Windows, Linux or Solaris. The mobile part is written in C++. A *runtime environment* establishes the underlying task structure and provides a front-end for configuring and controlling the system. It is divided into eight so-called *managers*. Each manager runs independently in the background and performs a specific task. E.g., the *Session Manager* handles session profiles, starts and stops sessions and supports joining and leaving sessions. The *Connection Manager* is active during a session and handles the communication between shared applications. It provides multicast mechanisms for information distribution between participating sites. The *Rendezvous Manager* offers services for activities before a session begins, including, e.g., session announcement to other team members.

Fig. 1 shows the stationary DreamTeam runtime stack with two sites. In reality, we often have sessions with more sites. To simplify this figure, we only show one groupware application. Usually, more than one application runs in a collaborative session simultaneously. We distinguish three levels of communication:

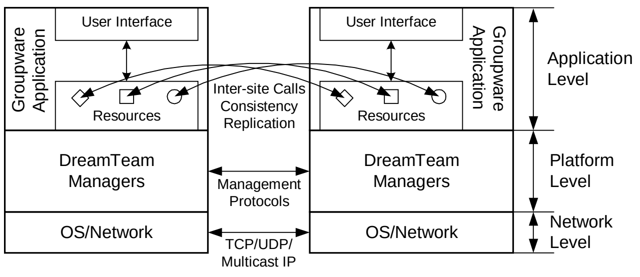


Fig. 1. The DreamTeam runtime and communication stack (stationary version)

- On *network level*, DreamTeam uses the Internet protocols TCP, UDP and, if available, native multicast with Multicast IP. The latter provides better scalability if a session contains a large number of communicating sites. Since Multicast IP does not provide reliable data transport, we integrated a reliable multicast layer into the platform.
- On *platform level,* each manager runs its own protocol. The most important protocols are the rendezvous protocol, the session management protocol, the protocol for member registration and the resource management protocol.
- The *application level* is the only level, an application developer perceives. On this level, we find state replication and consistency management as described in the next section.

3.1 Replication in a Stationary Environment

DreamTeam is based on a fully replicated communication infrastructure. Each site involved in a collaborative session is logically connected to each other site and runs an own instance of the shared application.

Distributed applications are built up of so-called *resources*. Resources are the shared building blocks of an application, e.g. shared texts, shared diagram elements, shared web pages or shared slides in a slide presentation tool. Resources can communicate with their corresponding peer resources by so-called *inter-site calls* – method calls which are synchronously executed on all participating sites. E.g. an inter-site call

anyResource.anyMethod(param$_1$, ..., param$_n$);

executes the method `anyMethod` on all replicated resource instances of `anyResource` in the session. To distinguish inter-site calls from local calls, the developer has to enter a specific keyword in the program code. Inter-site calls can roughly be compared to the *remote method invocation* (RMI) concept of Java. As a major difference, inter-site calls are sent to more than one site. In addition, the developer has not to manually load remote instances, since the runtime system builds up the identical resource structure on all sites automatically. The replicated structure makes all resources available locally. Inside a site, standard programming mechanism can be used without the need for an additional communication layer. This leads to efficient and straightforward application structures.

We first discuss the simple replication with only stationary users. To ensure consistency of concurrent updates, the runtime system requests pessimistic locks for each inter-site call without any developer's intervention. DreamTeam uses distributed locks as introduced by Suzuki and Kasami [18]. If an application changes a resource `R` by means of an inter-site call `I(R)`, the runtime system performs the following operations:

```
request(L(R))
apply I(R) to resource R
multicast I(R) to other sites
release(L(R))
```

Here, `L(R)` denotes the lock associated to R. The `request` statement blocks, when the lock is currently in use. This is acceptable in the stationary DreamTeam environment, where connections are reliable and fast. Thus, delays caused by locks are usually very short. The developer can override this generic scheme to increase parallel execution capabilities if consistency can be relaxed.

3.2 Replication in a Mobile Environment

Finding an appropriate architecture for the mobile extension is of central importance. To have real-time applications and weakly connected devices at the same time results in conflicting requirements: on one hand, updates of resource states should be distributed to all participants in real-time. On the other hand, disconnections are unavoidable in wireless networks. Often, handheld devices are simply disconnected because of auto-power-off services carried out by the operating system to safe battery power.

We resolve this conflict with the help of a design pattern called the *remote proxy pattern* [12]: the mobile device does not connect directly to other sites, but asks another computer, called the *proxy*, to act as a placeholder. The proxy performs heavy-duty tasks and stores data when the mobile device goes off-line. The idea of proxy pattern in general is not new. The first proxy pattern designed to describe networked applications was introduced by Shapiro [15]. His concept contains a *client*, a *service*, which the client wants to use across a network, and a *proxy*, which mediates between client and service. A similar idea was presented by Silva et al. [16]. Their proxy, called the *distributed proxy*, has a very fine-grained definition, which divides a system into *client*, *server*, *client proxy*, *server proxy*, *client communicator* and *server communicator*. This fine-grained definition is too specific for our intended domain, since only few systems meet this architecture in reality.

Shapiro's and Silva's proxies conceptually differ from the remote proxy pattern. As a major difference, the remote proxy pattern assigns the client and proxy processes to different computers. This offers the required flexibility to solve our problem.

Fig. 2 shows the resulting architecture. Components on the right side, i.e. existing DreamTeam installations, remain unmodified. The protocols on network, platform and application levels of stationary sites are identical to proxy protocols. Thus, DreamTeam systems can, from the viewpoint of communication, not distinguish mobile from stationary users. This saves implementation costs, since the proxy can use most of the DreamTeam managers without modification.

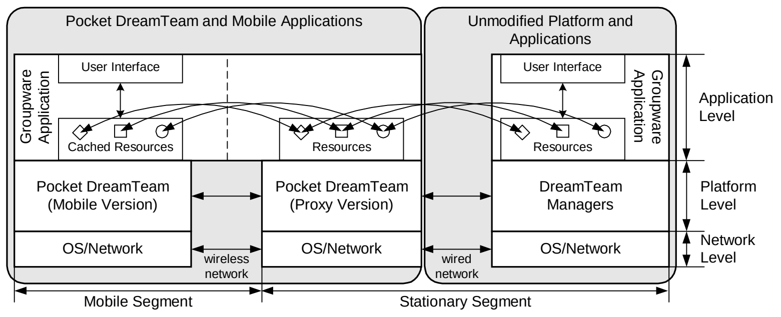


Fig. 2. The extended DreamTeam architecture with mobility support

The network is divided into a *mobile* and a *stationary segment*. A mobile groupware application has parts running on both segments. This seems to be a high burden, as parts of one application instance have to communicate across the network. We will see in a later section that the runtime system carries out most of the required communication services automatically.

Handheld users request low response time, for e.g. screen updates, even if the underlying wireless network causes high latencies. As a rule, handheld applications have to process user events in less than one second [1]. Thus, the mobile device stores for each resource a corresponding *cached resource*. The runtime system automatically updates cached resources when the original resources change their state.

3.3 Joining a Session

When a mobile user wants to join a session, the system has to perform three steps:

1. In the first step, a mobile device has to look up a proxy. A groupware infrastructure may have an arbitrary number of proxies running in stand-by mode. The proxy discovery protocol uses broadcast to ask all computers in a network whether they offer the proxy service. If the network supports DHCP (Dynamic Host Configuration Protocol), we can add the proxy address to the DHCP service record, which is passed to a mobile device when it enters a new subnet.
2. The proxy performs a *group rendezvous* [13], i.e. it looks up other DreamTeam sites that are currently on-line. Note, that there is no central server where DreamTeam sites are registered, thus the rendezvous protocol has to run completely decentralised.
3. The Session Manager on the proxy performs a *join* operation [10]. This operation copies the current resource structure from another participant and loads the current state information. Since other members change resources concurrently during the join operation, a complex protocol has to avoid race conditions.

3.4 The Role of the Proxy

In this architecture, a proxy computer has an important role, thus we have to discuss problems related to disabled or disconnected proxies. There exist three variations of this problem:

The proxy disconnects from other session members, but still has contact to the mobile device: In this case, the stationary network is partitioned into two or more segments. The proxy automatically performs a *leave* operation. The mobile user cannot collaborate with other session members, but can rejoin as soon as the interruption ends (steps 2 and 3 in section 3.3).

The proxy disconnects from the mobile user, but still has contact to the stationary segment: This case is more likely, since the mobile segment is much more prone for disconnections than the stationary segment. Other session members can continue without interruption. The proxy keeps track of all shared state changes, thus the mobile user can continue immediately after re-connection. During disconnection, a user can continue her or his work on the cached resources. Inter-site calls cannot be performed directly, as a network connection is a prerequisite for modifying a shared state. They have to either return an error or have to be queued up. In principle, it is possible to pass an arbitrary number of queued inter-site calls to the proxy after reconnection. Too large queues however may confuse users, thus the application developer can limit the size of the inter-site queue. Note that during disconnection other users may modify the shared state. Nevertheless, the consistency concept (see next section) preserves consistency of the shared state.

If a mobile user is disconnected longer than a certain time (e.g. some minutes), a *leave* operation is performed. In this case, the mobile user can look up another proxy inside the network and rejoin (steps 1 to 3 in section 3.3)

The proxy disconnects from both segments or breaks down: This is a combination of the cases above. Since application states stored inside the resources are replicated, no information gets lost.

3.5 Consistency

Obtaining consistency of shared data is a crucial point in weakly connected systems. Having loosely connected devices, we cannot solely use pessimistic locks any longer. A mobile device holding a lock could be disconnected for a certain time, thus other members would be blocked for a long time.

Pocket DreamTeam provides a hybrid approach for concurrency control: we use pessimistic locks for the stationary segment and optimistic conflict detection and resolution [4, 19] for the mobile segment. For this, the proxy contains two threads, which wait for incoming messages. The first thread waits for inter-site calls from other participants (i.e. all session members apart from the associated mobile member). We outline the thread as follows:

```
do {
   receive inter-site call I(R) from other site
   // note: other site already requested the lock
   apply I(R) to local resource R
   increase T(R)
   if (mobile device is on-line)
```

```
         send state of R and T(R) to mobile device
      else
         store state and T(R) and send when device re-connects
   } until (session stops)
```

This thread updates the local resource state continuously. In addition, it copies new states to the mobile device, which stores them in its cache. `T(R)` denotes a logical timestamp used in the second thread for conflict detection. The second thread waits for messages from the mobile device:

```
   do {
      receive inter-site call I(R) and T'(R) from mob. device
      request(L(R))
      if (T'(R)<>T(R))  // i.e. conflict!
          solve conflict
          // i.e. generate new I(R) without conflicts
      apply I(R) to resource R
      increase T(R)
      send state of R and T(R) to mobile device
      multicast inter-site call I(R) to other sites
      release(L(R))
   } until (session stops)
```

To detect conflicts, the mobile device sends in addition to `I(R)` the logical timestamp `T'(R)` of the last cache copy of `R`. This allows the proxy to detect easily, whether `I(R)` is associated to an older copy of `R`. In this case, the proxy has to perform a conflict resolution. Two generic conflict resolution strategies are:

- I_{new}`(R):=I(R)`, i.e. the mobile device has priority,
- I_{new}`(R):=null operation`, i.e. other members have priority. This is the generic way for Pocket DreamTeam to resolve conflicts.

Sometimes, a more fine-grained resolution strategy suits better. An optimal strategy takes into account the state of `R`, which the mobile device has perceived *before* it applied `I(R)`. We can get this state in two ways:

- The mobile device sends in addition to `I(R)` and `T'(R)` the old state of `R`. This however increases network traffic.
- The proxy stores old states in a hash table and uses `T(R)` as key. Whenever it receives a new timestamp `T'(R)`, it can remove older entries.

With this extension, we could implement even complex consistency algorithms based on *operational transformation* [2]. Note that the consistency mechanism realised inside the platform ensures data consistency on a basic level. An application developer however is free to implement high-level strategies based on e.g. social protocols. A shared text editor can, e.g., offer functions to reserve text paragraphs for exclusive editing. Such strategies are highly application-dependent and not part of the platform. Nevertheless, they require low-level consistency of data as provided by our consistency mechanism.

3.6 Implementation Issues

Realising software for handheld devices is hard work, since developers have to deal with small memories, slow processors and restricted operating system capabilities. Pocket DreamTeam requires portions of code on the handheld under C++ as well as portions under Java. One design goal of Pocket DreamTeam was to heavily re-use stationary portions in mobile applications. In order to create a Pocket DreamTeam application, we have to perform four steps:

1. For the proxy portion, we can copy resources from the stationary version. We have to add code for marshalling/unmarshalling the state, since state information is transferred across language borders (Java to C++).
2. We have to add code for conflict resolution, if the generic strategy is not suitable.
3. For the mobile portion, we have to transfer parts of the resource code to the target platform (i.e. C++). As a minimum we have to code all data fields. The runtime system automatically passes inter-site calls to the proxy. The proxy in turn performs the appropriate state change and sends the new resource state back. This can cause long turn-around times. As a solution, we can transfer time-critical inter-site calls to the mobile device, which then modify cached state directly.
4. We implement the rest of the mobile portion, especially the user interface, like a single user application without considering communication or replication issues.

3.7 Testing Environment and Sample Applications

We completely implemented and tested Pocket DreamTeam. As a technical platform for mobile end-user devices we use handhelds as shown in fig. 3. We in particular decided not to use notebooks. Due to their size, weight and battery life, mobile working capabilities with notebooks are limited.

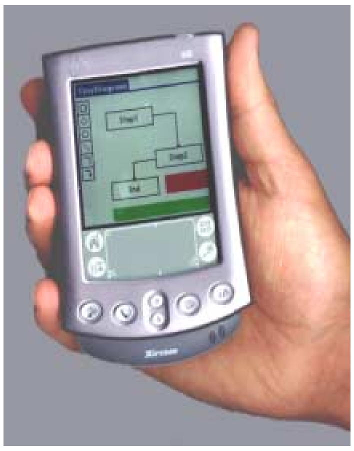


Fig. 3. End-user device with Pocket DreamTeam running a collaborative diagram application

Our development and testing system consists of

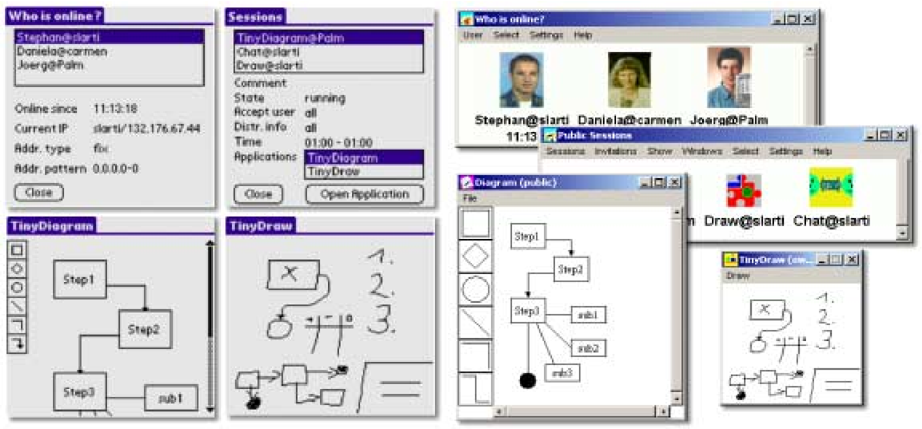
- two handheld devices (Palm m505 with PalmOS 4.0) equipped with wireless LAN (IEEE 802.11b) adapters,
- a number of stationary workstations (Windows PCs, Solaris workstations),
- a wireless LAN infrastructure connected with the campus Internet.

Although our testing environment primarily bases on wireless LAN, we strictly paid attention to be as independent as possible of the network. In principle, Pocket Dream-

Team could run on other wireless networks such as IrDA (Infrared), Bluetooth or GSM. Since not all networks support the Internet Protocol (IP) sufficiently, we isolated network related functions in a component we call *Network Kernel Framework* (*NKF*). NKF can roughly be compared to a network driver and offers a uniform interface to higher communication layers.

As mentioned above, software for the end-user devices were coded in C++. Even though Java would fit much better into the overall system architecture, the handheld version (*Java Micro Edition*) was not capable enough for our project. We decided to use the well-established development environment *CodeWarrior* for PalmOS.

To test the concept, we implemented the DreamTeam core applications as well as two groupware applications on top of Pocket DreamTeam.


Stephan@slarti
Daniela@carmen
Joerg@Palm

Online since 11:13:18
Current IP slarti/132.176.67.44
Addr. type fix
Addr. pattern 0.0.0.0-0

TinyDiagram@Palm
Chat@slarti
Draw@slarti

Comment
State running
Accept user all
Distr. info all
Time 01:00 - 01:00
Applications TinyDiagram, TinyDraw

Fig. 4. Groupware applications on mobile device (left) and desktop (right)

Fig. 4 shows Pocket DreamTeam windows (left) and the corresponding desktop DreamTeam windows (right). Corresponding screens on different platforms may look completely different. Overlapping windows, context menus and icons are not useful on small screens. E.g., we replaced icon-based dialogs by simple textual lists.

The users use the upper frames to view contextual information and control sessions. The *On-line list* shows all users, which are currently on-line, i.e. can participate in a collaborative session. The *Sessions* frame shows all running and planned sessions. A user can select a running session from a list and join. From the 20 DreamTeam applications, we selected two applications for mobile extension:

- The *Diagram* tool allows a team to collaboratively create diagrams such as flow charts, entity relation ship or class diagrams.
- With the *Draw* tool, a group can draw and share simple free-hand sketches.

To realise these applications, we performed the steps as described in section 3.5:

1. For e.g. a diagram resource (i.e. a rectangle or a circle) we had to enter code to marshall the co-ordinates, size, colour etc.
2. *Draw* as a very simple application uses the generic resolution method. *Diagram* is more complex. Conflicts occur, when two users modify the same resource simultaneously. Our conflict resolution method first investigates, whether the modification

affects the same data fields. If not, we apply both modifications, since no real conflict occurs. Otherwise, we branch to the generic conflict resolution.

3. To increase performance of the *Draw* application, we transferred the method, which adds a line to the sketch to the mobile device. This significantly improves response time during free-hand drawing.
4. We implemented all dialog frames, menus, buttons etc. inside the target environment.

Table 1. Comparison of DreamTeam implementations

	Stationary	**Proxy**	**Re-used**	**Mobile**
Core platform binary	2.3 MB	2.0 MB	-	72 KB
Core platform source	125000 lines	110000 lines	98 %	9500 lines
Diagram source	6900 lines	5200 lines	92 %	600 lines
Draw source	930 lines	520 lines	90 %	410 lines

Table 1 summarises our implementation efforts. This table compares the implementation efforts for stationary DreamTeam and Pocket DreamTeam (proxy and mobile portions). The column *Re-used* indicates how much source code of the stationary version could be re-used in the proxy. We can see a high degree of re-usable source code for the proxy implementation. In addition, the required source codes for mobile portions are considerable small. Especially the platform core binary of 72 KB demonstrates that Pocket DreamTeam is suitable for devices with small memories.

4 Conclusion and Future Work

Pocket DreamTeam demonstrates how we could effectively extend a distributed application platform for mobile usage. We used the remote proxy pattern as a guideline for our architecture. Mobile users can access high-demanding applications through devices with low computational power. The software architecture based on resources dramatically simplifies the implementation, since the runtime system is able to carry out most of the required communication and replication services automatically. Drawbacks related to the remote proxy pattern (e.g., problems with disabled proxies) are addressed by higher-level mechanisms. For replication and consistency, the system offers generic mechanisms, which a developer can adapt. Especially the combination of pessimistic concurrency control for the stationary segment and optimistic concurrency control on the mobile segment is unique and combines the advantages of two concurrency control concepts.

In the future, we want to reduce the implementation efforts for mobile applications even more. For this, we plan to develop a program, which generates source code for proxy resources and mobile cached resources automatically from stationary resources. This however requires some syntax extensions, e.g. new keywords, but would significantly reduce development costs.

References

1 Bey C., Freeman E., Hillerson G., Ostrem J., Rodriguez R., Wilson G., Dugger M.: Palm OS Programmer's Companion, Volume I, Palm Inc, July 2001
2. Cormack G. V.: A Calculus for Concurrent Update, Department of Computer Science, University of Waterloo, Waterloo, Canada, 1995
3. Joseph A. D., Tauber J. A., Kaashoek M. F.: Mobile Computing with the Rover Toolkit, IEEE Transactions on Computers, Vol. 46, No. 3, March 1997, 337-352
4. Kistler J. J., Satyanarayana M.: Disconnected Operation in the Coda File System, ACM Transaction on Computer Systems, Vol. 10, No. 1, Feb. 1992, 3-25
5. Lotus Development Corporation: Lotus Notes, http://www.lotus.com/home.nsf/welcome/developernetwork
6. Lukosch S., Roth J.: Reusing Single-user Applications to Create Multi-user Internet Applications, Innovative Internet Computing Systems (I2CS), Ilmenau, June 21-22, 2001, LNCS 2060, Springer, 79-90
7. Munson J. P., Dewan P.: Sync: A Java Framework for Mobile Collaborative Applications, special issue on Executable Content in Java, IEEE Computer, 1997, 59-66
8. Myers B. A., Stiel H., Gargiulo R.: Collaboration Using Multiple PDAs Connected to a PC, Proceedings of the ACM 1998 conference on Computer supported cooperative work, 1998, 285-294
9. Noble B., Satyanarayanan M., Narayanan D., Tilton J. E., Flinn J., Walker K.: Agile Application-Aware Adaptation for Mobility, Proceedings of the 16th ACM Symposium on Operating System Principles, Oct. 1997, St. Malo, France
10. Roth J.: DreamTeam - A Platform for Synchronous Collaborative Applications, AI & Society (2000), Vol. 14, No. 1, Special Issue on Computer-Supported Cooperative Work, Springer London, March 2000, 98-119
11. Roth J.: Information sharing with handheld appliances, 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01), Toronto, Canada, May 11-13, 2001, LNCS 2254, Springer, 263-279
12. Roth J.: Patterns of mobile interaction, Proceedings of Mobile HCI 2001: Third International Workshop on Human Computer Interaction with Mobile Devices, M. D. Dunlop and S. A. Brewster (eds), IHM-HCI 2001 Lille, France, Sept. 10, 2001, 53-58
13. Roth J., Unger C. : Group Rendezvous in a Synchronous, Collaborative Environment, in R. Steinmetz (ed): Kommunikation in Verteilten Systemen (KiVS'99), 11. ITG/VDE Fachtagung, 2.-5. March 1999, Springer, 114-127
14. Roth J., Unger, C.: Using handheld devices in synchronous collaborative scenarios, Personal and Ubiquitous Computing, Vol. 5, Issue 4, Springer London, Dec. 2001, 243-252
15. Shapriro M.: Structure and Encapsulation in Distributed Systems: the Proxy Principle, Proc. of the 6th Internal. Conference on Distributed Computing Systems, Mai 1986, 198-204
16. Silva A. R., Rosa F. A., Gonçalves T., Antunes M.: Distributed Proxy: A Design Pattern for the Incremental Development of Distributed Applications, Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), Davis, November 2000, California, USA, LNCS 1999, Springer, 165-181
18. Suzuki I., Kasami T.: A distributed mutal exclusion algorithm, ACM Transactions on Computer Systems, Vol. 3, No. 4, Nov. 1985, 344-349
19. Terry D. B., Theimer M. M., Petersen K., Demers A. J.: Managing Update Conflict in Bayou, a Weakly Connected Replicated Storage System, Proceedings of the fifteenth ACM symposium on Operating systems principles, Copper Mountain, CO USA, Dec. 3-6, 1995, 172-182