Supporting Collaboration in Heterogeneous Environments

ALLAN M. KREBS, BOGDAN DOROHONCEANU, AND IVAN MARSIC

ALLAN M. KREBS is a Research Associate at Rutgers University, CAIP Center. He received his M.S. in Electrical Engineering from the Technical University of Denmark. His research interests are within the fields of groupware, mobile computing, distributed software objects, distributed computing, and software engineering. He is the chief architect of the DISCIPLE middleware.

BOGDAN DOROHONCEANU is a Ph.D. candidate in computer science at Rutgers University. He did his undergraduate studies at Politehnica University of Bucharest, Romania. Bogdan's research interests are mainly in groupware architectures, multiuser interfaces for groupware, software engineering, and Java technologies. He was a summer intern with IBM Almaden Research (San Jose, CA) and Siemens Research, Inc. (Princeton, NJ).

IVAN MARSIC is an Associate Professor in the Department of Electrical and Computer Engineering, Rutgers University. He earned his B.S. and M.S. from University of Zagreb, Croatia, and his Ph.D. from Rutgers University. His research interests include computer networks, groupware, mobile computing, distributed software objects and agents, software engineering, and multimedia.

ABSTRACT: Heterogeneous sharing in synchronous collaboration is important with the proliferation of diverse computing environments, such as wearable computers and handheld devices. We present here a data-centric design for synchronous collaboration of users with heterogeneous computing platforms. Our approach allows clients with different capabilities to share different subsets of data in order to conserve communication bandwidth. We have built a robust middleware consisting of a distributed repository of shared data objects and a client-server-based infrastructure. Using the middleware, we have developed a framework for building collaborative applications for clients with different display and processing capabilities. We discuss the design and implementation of our middleware and framework and evaluate them by building four complex sample applications that demonstrate scalability, good performance, and high degree of code reusability.

KEY WORDS AND PHRASES: collaboration, heterogeneity, information systems, middleware, mobile devices.

REAL-TIME SHARING AND MANIPULATION of information using very different display capabilities on the diverse devices are key components for synchronous collaboration.

The old problem of adapting the user interfaces of the applications to multiple platforms has become even more important with the large diversity of mobile connected devices that has emerged within the past couple of years. The use of various Web browsers gives a partial solution to the problem, but if the needed application is much more complicated than manageable using HTML forms, something better needs to be developed. Other problems occur when the applications become collaborative, especially when the users are using heterogeneous devices that can interoperate using common standards-conforming hardware and software interfaces.

Most collaborative applications provide synchronization support through the "what you see is what I see" (WYSIWIS) technique [25]. WYSIWIS allows one user to point or discuss an object on the screen that they are confident is visible to their collaborator. Providing support for WYSIWIS with past systems was a difficult, but not impossible, task because the groupware used common platforms. Developing groupware applications that are interoperable across diverse environments is significantly more difficult and costly.

However, with the recent anytime-anywhere proliferation of computing technology, support for heterogeneity is inevitable. Mobile applications involving synchronous collaboration are emerging in many fields, such as business, health care, military services, and transportation. The classic example is the provision of just-in-time assistance between a deskbound expert and a mobile fieldworker using a portable device. The mobile worker may work with blueprints, whereas the expert is using a three-dimensional CAD model to repair a vehicle or the plumbing in a building. Kraut et al. [15], for example, show that fieldworkers make quicker and more accurate repairs when a remote expert is providing assistance.

The heterogeneity of computing platforms has dimensions of processing speed, storage capacity, input devices, display capabilities, response latencies, network reliability, and bandwidth, with network and display accounting for the most prominent differences. This paper focuses on different display capabilities and on network reliability and bandwidth. We propose a solution for environments where clients have (1) slow or unreliable connections, such as wireless connection to the Internet; (2) different display and processing capabilities, such as PDA versus desktop workstations; and (3) data that can be distributed over multiple servers in the presence of slow or unreliable connections. These problems appear in areas that often involve mobile devices connected using low bandwidth links such as military or civilian rescue or disaster relief applications for distributed operations planning. Another area is business applications with mobile users, where the users need to collaborate on shared resources, such as a corporate calendar and a resource management system.

The solution proposed here is a framework for building collaborative applications for a heterogeneous environment. We use software agents to synchronize and distribute the data objects and to dynamically load code for transformation of data between different representations. The clients can run on a variety of platforms including desktops, Web browsers, and portable or handheld devices.

Related Work

THE NEED TO ALLOW CONFEREES TO COLLABORATE on dissimilar terminals was recognized early on by Engelbart [4], the pioneer of computer-supported collaborative work. An early design for heterogeneous groupware is presented in Karsenty et al. [11], but it does not deal with platforms of significantly different computing and communication capabilities. Rendezvous [8], GroupKit [23], and several groupware toolkits thereafter use model-view separation so that developers can create models and drive different views. However, no such implementation is reported.

The Visage system from Maya Design [7] is a powerful (single-user) visualization system for creating custom visualizations and direct manipulation of large and diverse data sets. Although Visage addresses diverse data visualization with polymorphic views, it is not explicitly intended for collaboration in heterogeneous computing environments. Recent work on Visage Link [19], an extension of Visage, adds multiuser collaboration capabilities, but Visage Link does not consider heterogeneous models.

A system for data management support is presented in Preguiça et al. [22]. The main focus is on asynchronous groupware and integrating awareness support, but it is related with our work, as it uses a replicated object store and local caching to support mobile clients. It differs mainly by having an application-dependent distributed object (called Coobject), which manages the data associated with a specific application. Our approach has a clearer separation between data and application, which simplifies development of different applications sharing the same data objects. Also, having more lightweight data objects makes easier implementation on small devices, such as PDAs. Preguiça et al. [22] does not address the different issues specific for synchronous groupware or heterogeneity of clients.

An important aspect of heterogeneity is interoperability between the existing groupware systems that use different group coordination policies or different software architectures. Dewan and Sharma [2] address this issue. Conversely, we address the case where the participants' platform capabilities are significantly different.

Although the above-mentioned related work addresses many issues in collaborative systems and groupware, most of them are not concerned with heterogeneity and none with the special cases for wireless connected small clients. Our essential principles for heterogeneous collaboration are that every user's action is interactively and continuously reflected in other users' workspaces, with a varying degree of accuracy or realism, or through a qualitatively different visualization, and that users should be able to work off-line and get their work seamlessly synchronized with that of the groups, upon reconnect.

More recently, the need for middleware to support collaborative groupware applications have been identified as a core component in groupware systems. Many middleware systems provide some sort of event filtering mechanism and often the events to be filtered are represented in Extended Markup Language (XML) (e.g., CORTEX, STEAM, XMIDDLE, Proem, DACIA, Hermes, and SIENA).

CORTEX [3] is based on anonymous and asynchronous event models. CORTEX is a context-aware middleware architecture that provides support for both pervasive and ad hoc computing. The middleware is structured by a number of components, a publish–subscribe component, a context component, a service discovery component, and a resource management component. The CORTEX event model uses XML to represent events. The dissemination of events over the network is achieved using SOAP Messaging [24]. CORTEX uses STEAM [20], which is an event-based middleware service intended for wireless LANs in ad hoc mode, where events are a type–value pair. STEAM uses three types of event filters: subject, filtering events according to consumer interest; proximity, filtering according to consumer location within the network; and content, filtering according to the event value. The former two are located at the producer side, and the latter at consumer side. Unlike our system, STEAM does not address the issues of context filtering.

XMIDDLE [18] is a mobile computing middleware intended for wireless peer-to-peer networks in ad hoc mode. Like our system, XMIDDLE uses XML to represent data and XML Document Object Models (DOM) (trees) to manipulate data, allowing online collaboration on shared data DOM objects and off-line manipulation and synchronization. XMIDDLE focuses on how XML DOM objects are shared, protocols for reconnection and data reconciliation, after off-line manipulation of data. The XPath [31] and XLink [30] languages are used to format linking and addressing of XML nodes in a tree and reference remote trees. IBM's XML TreeDiff [9] is used for tree reconciliation and off-line modification. XMIDDLE does not implement any filtering of events or provide resource management.

Proem [13] is a middleware platform for developing and deploying peer-to-peer applications for mobile ad hoc networks. The system is built up around peerlets, individual peer-to-peer applications that use the publish–subscribe event model. Currently, the Proem system consists of a peerlet engine, a run-time system, and a development kit. The peerlet engine instantiates, executes, and terminates peerlets. The run-time system contains three main entities: the peerlet engine, a set of basic services, and a protocol stack. The basic services are a set of managers handling peer-to-peer services (announcements, discovery of peers, keeping track of peer community memberships, a peer database, a log of encountered peers), a data space manager for persistent storage of data spaces and access control and an event bus manager, enabling event communication among peers using the publish–subscribe event model. Each service is implemented as a peerlet. The protocol stack consists of a presence protocol, containing messages that announce a peer's presence in the network; a data protocol, containing data messages; and a community protocol, containing messages for applying, granting, and verifying community membership. As in our system, events are represented in XML.

DACIA [12] is a framework for building adaptive distributed mobile groupware applications that adapt to available resources, support user mobility, and dynamic reconfiguration. The architecture consists of three core elements, processing and routing components (PROCs), a resource monitor, and a PROC execution engine. PROCs are the main components in an application, being either data objects or executables.

The resource monitor monitors the local PROCs performance as well as making reconfiguration decisions, notifying the engine of such reconfigurations. It may also provide application specific policies and is an optional component. The engine is responsible for maintaining a list of PROCs and their connections, establishing and maintaining connections between hosts and communication between hosts. Currently, one engine is needed on every host. The focus of DACIA is support for application and user mobility, enabling users to move an application from one computing device to another, dynamic application reconfiguration, such as dynamically loading new components or changing the way various components interact and communicate and application parking, allowing disconnected users to continue to interact with some limitations.

Hermes [21] is a system that focuses on providing a scalable event-based middleware for large-scale distributed applications. Hermes proposes to create an event dissemination tree by employing a routing network that manages rendezvous nodes for advertisements and subscriptions in the network. Event filtering is done at broker nodes and rendezvous nodes. Rendezvous nodes are initially set up by publishers who send a "type message," indicating that a new event type is introduced. The rendezvous node is the broker-node whose ID is the hash of the new event type name. The consumers then subscribe to events at the rendezvous node. Every broker-node along the path from the publisher or subscriber keeps state of the event forwarded. When an event is published, it is routed up to the rendezvous node, filtered and the event follows the reverse path to all interested subscribers. Publishers and consumers connect locally to an event broker-node to send and receive events. Global communication is done through the broker-nodes and rendezvous nodes.

As mentioned, Hermes is intended for large-scale distributed systems where network resources are abundant and fixed; hence, filtering according to resource availability is not an issue in Hermes.

SIENA [1] is an event notification service, intended to be accessible from every site in a wide area network. The focus of their research is maximizing expressiveness in the filtering events without affecting negatively on scalability of the delivery system. It is implemented as a distributed network of servers so as to achieve a scale suitable for large numbers of clients and high volumes of notifications spread across a wide area network. In current research, SIENA is being extended with focus on wireless mobile publish–subscribe applications.

Another approach to real-time collaboration is to specify a protocol that supports collaboration across multiple platforms. The Web Distributed Authoring and Versioning (WebDAV) [29] working group of the Internet Engineering Task Force (IETF) has adopted this protocol-centric approach, and developed a novel network protocol, the WebDAV Distributed Authoring Protocol, which supports interoperable remote, asynchronous, collaborative authoring. The WebDAV protocol is a set of extensions to the HTTP protocol, which provides facilities for concurrency control, name-space operations, and property management. The protocol allows users to collaboratively author their content directly to an HTTP server, allowing the Web to be viewed not just as a read-only way to download information but as a writable, collaborative medium.

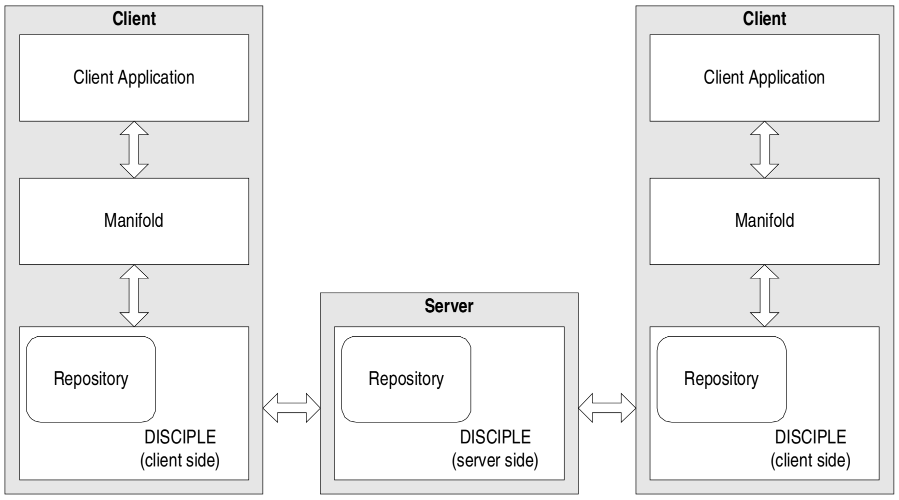

Figure 1. Overview of the Collaborative System Architecture.

WebDAV focus is on sharing of files, whereas our effort targets a much finer granularity with XML attributes as the lowest level object to share. WebDAV is supported by a number of document-authoring applications, file managers, and servers, such as MS Word 2000, MS Excel 2000, Adobe Acrobat 5, GNOME Nautilus, MS IIS 5, Oracle Internet File System, and WC3 Jigsaw.

System Architecture

THE SYSTEM ARCHITECTURE of the collaborative framework comprises three main layers (see Figure 1): (1) DISCIPLE middleware for data sharing (distributed repositories and information bus for basic communication), (2) manifold framework for developing collaborative applications for heterogeneous environments, and (3) task-specific applications.

Data Sharing Middleware

The data sharing middleware consists of synchronizable distributed repositories and an information bus.

The role of a repository is to store information persistently. Repositories are one of the main components of the system, and are distributed and partially replicated over both servers and clients. The main idea of the repository is that each client has a copy (and, in some cases, the only copy) of the objects that the client is interested in. In the example of a disaster relief application, the objects could be all objects that are physically within a ten mile range. Almost the same rule applies to the servers: the servers hold a copy of the objects that their clients are interested in. Updates to the objects are propagated between all the entities that hold a copy of the same object. The approach for dealing with this propagation and with the possible conflicts is described below.

Listing 1. The UForm Interface.

```
  public interface UForm {
    public void setId(long id);
    public long getId();
    public boolean isDirty();
    public void setDirty(boolean dirty);
    public boolean isDeleted();
    public void setDeleted(boolean deleted);
    public void setType(String type);
    public String getType();
    public boolean hasProperty(String key);
    public Object getProperty(String key);
    public void setProperty(String key, Object value);
    public Object removeProperty(String key);
}
```

The main goal of the server and repository design is to make the server and repository completely independent of the application semantics. We have built two implementations for the communication infrastructure.

Earlier Version

In an earlier implementation of the communication infrastructure, this is achieved by using an abstract data type, called UForm (short for "universal form"), which encapsulates the data [16]. The UForm essentially consists of a unique identifier and a keyed list of properties. The interface for the UForm is shown in Listing 1. Thus, the repository separates data from presentation, as in model-view-controller, but the separation is mediated via pure data entities (UForms), rather than via active objects. The content of the repository defines the application state.

In addition to the unique ID and the property list, an UForm contains additional fields used in merging the states when the client becomes temporarily disconnected from the server. For example, the *dirty* field specifies whether or not the UForm was modified while the client was off-line. When an object is deleted, it is not removed from the repository or from the client state, but instead is marked as deleted using the *deleted* field.

The operations on a repository are defined in a simple Java interface, which is a common interface for both servers and clients. The implementation is not bound to any specific data structure. It could be anything from a hash table to an object-oriented database, and the implementations can be different across the clients and the servers. In the present reference implementation, we have used the hash table, both on servers and clients.

The UForms can be linked in complex data structures by using UForm identifiers as properties in other UForms.

Updates of the data are based on the command design pattern [6] with a small number of commands in the core design. The core command hierarchy is shown in

Figure 2. We have chosen to keep the number of different commands close to a minimum to keep the implementation simple, having fewer commands to take into account.

The commands support implementations using the difference or delta mechanism as far as adding, removing, moving, and copying elements and attributes, but they do not support a difference mechanism where numbers, for example, are updated by sending relative increments in coordinates rather than sending absolute coordinates. It is only possible to update a text element or an attribute by overwriting it with a new value.

New services can add their own commands by extending one of the commands in the command hierarchy, and it is actually possible to develop a whole new command structure to replace the one provided, if another approach is desired.

The information bus allows applications that use different data sets to share the same server instance (session) and the same repository on the server. This is particularly important for systems with limited resources that want to work with more than one collaborative application at a time (e.g., a chat and a situation-map application).

Current Version

In a more general sense, the key role of middleware can be the provision of services for access and management of low-level resources. Thus, the problem we face is how to configure a variable number of services and provide easy access to them. The current version of the core service of the DISCIPLE middleware provides a distributed repository for sharing of XML documents. This core service is supported by a number of other services such as state merging, history, and event notification services.

The main issue of designing the architecture of the DISCIPLE middleware was the decision on what should be counted as part of the core, and therefore non-replaceable, and what should not, and therefore replaceable. The support for multiple communication architectures, such as client-server and peer-to-peer, was required, but the question was to what extent we could generalize the design to support different configurations for data model, data distribution/event notification, concurrency control, latecomer support and (re-)synchronization, all areas of research in our group, and therefore a target for the middleware as test bed. Answering the question about the extent of generalization is the key to determining the interfaces between the core components and the support (replaceable) components.

We decided to use the XML DOM as our basic data model. This does not imply that the internal data structure of an implementation of the repository has to be XML DOM but, rather, that this is the data model for the interfaces to the repository and that data being transferred will be accessible as XML DOM objects. The decision to use XML DOM as data model was based on the flexibility given in the XML DOM representation with respect to manipulating and traversing the document, and the fact that we, by using XML DOM, can handle any XML document. The application data does not have to comply to any XML Schema or Document Type Definition (DTD).

Alternative solutions could have been to have a flat data structure where the elements were stored as independent objects linked together by an external mechanism

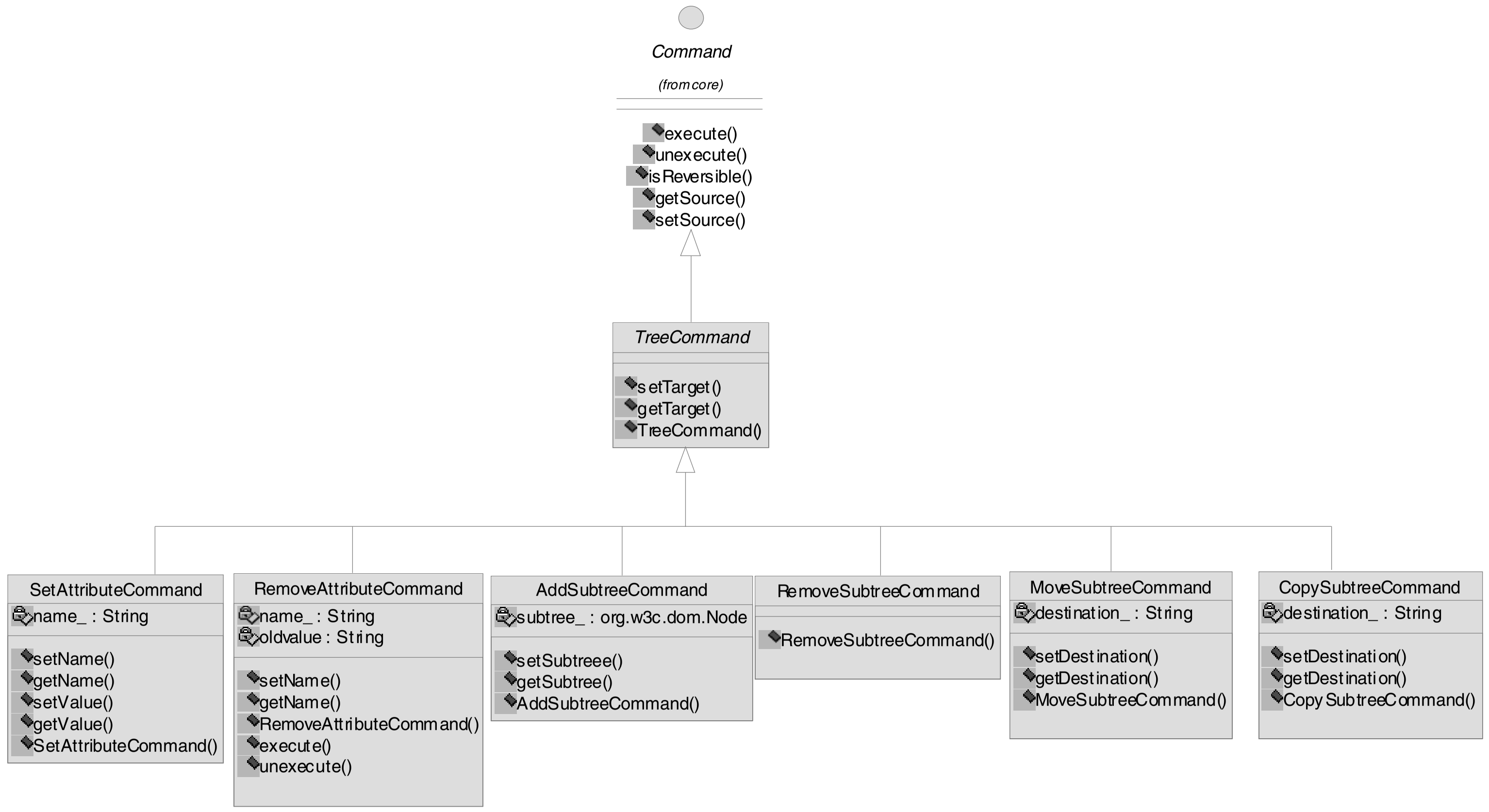

Figure 2. The Hierarchy of Command Classes of the Current DISCIPLE Middleware Version.

(as in the first implementation), or build our own tree structure parallel to the structure of the XML document. Both of these solutions require some sort of identification scheme for the objects to be able to recreate the structure of the original XML document; that is, restoring child–parent relationships. The identification has to be tagged to all the elements and has to be unique over all nodes, thus complicating the design unnecessarily. In our research on intelligent data distribution and event notification, we use rules to determine which events to forward to other peers. To decide if an event should be forwarded or not, these rules may not only use information from the specific element targeted in the event, but also from other elements in the tree, such as parents and siblings. This need for access to the whole document, rather than only a single element, also speaks for the use of the DOM.

Having decided on a data model, our next issue was on how updates of the data should be handled. We basically have two options: operation-based or content-based. Operation-based means that we send the operations through the middleware to the peer, which then perform them on the local data, whereas using the content-based approach, the updated data object is sent to the peer.

Given that the middleware is mainly targeted for mobile environments, wireless connected, the network resources will be limited. One of the important features of the infrastructure should therefore be to support configurations optimized for the limited network resources.

Both approaches have their advantages depending on the environment and application. Sending only the operations will, in most cases, use less bandwidth when in the middle of a session, but can be tedious when synchronizing after a (re-)connect. This is because the server or the other peers need to keep a log of all the operations performed, and then on synchronization send all the operations in the log to the new or reconnected peer. If many of the operations were performed on the same data object, many of them are probably obsolete, and sending the end-state of the object would be more efficient. For example, if you move a graphical object around, changing the coordinates, sending the operations for all the moves, will result in a higher quantity of data being transmitted to peers than just sending the final position.

On the other hand, sending the updated data object every time it has been changed can be less efficient than sending the operations when in a session. This depends on the size of the object needed to be sent, if only the exact field being changed is sent; it may not use more bandwidth than sending the operations.

We have chosen a combination of the two approaches by having the operations sent during a session, but sending the data objects—the content—upon (re-)connect, thus using the best from the two approaches.

Manifold Application Framework

To facilitate the development of collaborative applications for heterogeneous computing environments, we defined the Manifold framework for building collaborative applications. The framework consists of a base package with five Java interfaces for the basic structure of an application using the model-view-controller design pattern.

Default implementations for some of these interfaces have also been provided to help developing applications using the framework.

The basic interfaces are intended for use in all implementations of Manifold regardless of whether the target platform is a desktop workstation, a PDA, or a mobile phone. An application built with the framework can run as stand-alone or as an applet in a Web browser. The interfaces are simple enough to be used under the present version of J2ME CLDC [27] for Palm Pilot handheld computers. The basic interfaces are as follows:

- *Controller* is the command dispatcher and provides the interface for communicating with the application. It accepts the locally generated Commands and the remote Commands from peer clients, and executes the Commands on the local Repository. In the current version it is part of the DISCIPLE middleware.
- *Glyph* is the graphical representation of a node or sub-tree. The scene graph of an application may also contain polyglyphs (composite glyphs), which correspond to branch nodes and are containers for collections of glyphs.
- *Viewer* is the canvas or window, where the glyphs are displayed, such as icons on a map. The Viewer also handles user events.
- *Tool* encapsulates the current manipulation mode and defines the kinds of actions can be performed on the data objects or nodes. Typical tools include create tool, select tool, and property change tool.
- *Manipulator* does the actual manipulation in response to the user actions. It creates Commands in response to the manipulation and passes them to the Controller for execution.

Notice that we maintain copies of the shared XML documents in the repositories on the clients (Figure 2) in order to enable off-line work. The resulting model can be compared to the model-view-controller design pattern. The XML document corresponds to model and glyphs correspond to views/controllers. Figure 3 shows an example collaboration diagram, in this case for creating an object:

1. The user clicks on the application window to create an object. The mouse event is sent to the Viewer.
2. The Viewer invokes the method createManipulator() on the current Tool (which, in this example, is an object-creation tool). The Tool returns a Manipulator that encapsulates the manipulation sequence for creating a specific kind of object, such as a geometric figure.
3. The Viewer next invokes grasp() on the returned Manipulator. The Manipulator creates the node and possible sub-nodes and encapsulates it in a Command, which is returned to the Viewer.
4. The Viewer forwards the Command to the Controller.
5. The Controller sends the Command to the server through the DISCIPLE Client module. Then, depending on the concurrency strategy, it either waits for the Command to return back from the server or process it immediately (step 6).
6. The Controller invokes execute() on the Command.

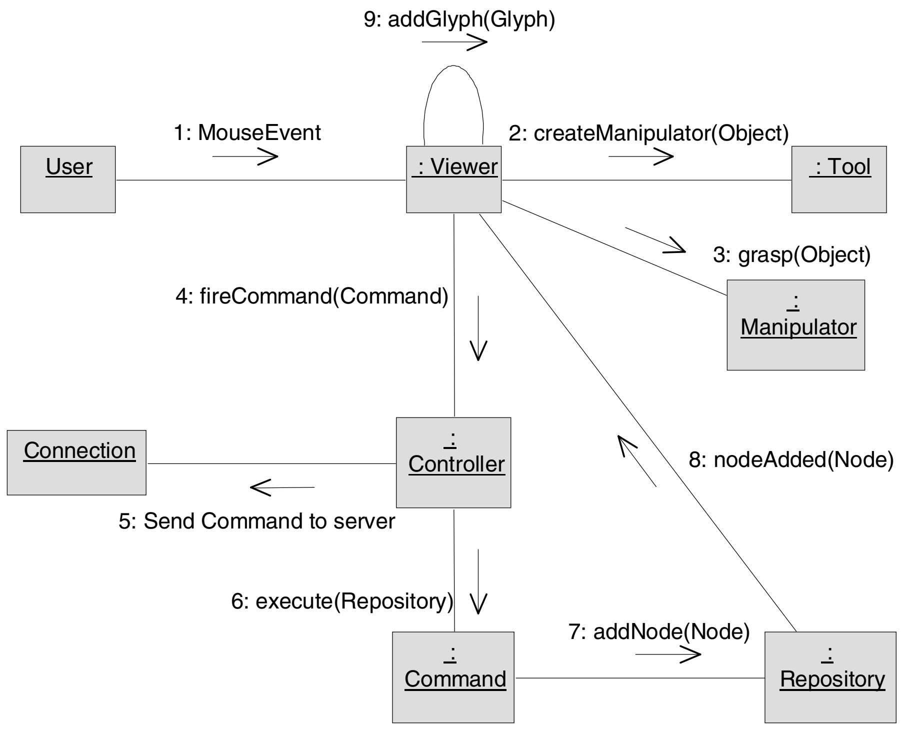

Figure 3. Collaboration Diagram for Creating a New Data Object.

7. This causes the Command to invoke addNode() on the local Repository, which adds the node to the Repository.
8. The Repository then informs the Viewer that an object has been added to the Repository. How this is done is not defined in the basic framework, and it is left to the application developer to define the actual mechanism.
9. The Viewer adds a view of the data object (a glyph) to its scene graph and links it together with the node. Again, this is not defined in the basic framework, but a reference implementation is provided.

A similar collaboration diagram applies for manipulating the properties of the data objects.

Distributed Repository Synchronization

THE IDEA BEHIND THE DISTRIBUTED REPOSITORIES is that data resides on the servers with clients being interested in the specific data objects. A client retrieves a copy of data objects into its own repository while working. Therefore, data is moved or copied around between the servers and clients.

Each user gets only a subset of shared data specified by the data distribution conditions. When a user first starts the collaborative application, a default condition is set for the user where the user receives all the data in the session. At run-time the user can change the conditions via a user interface offered by the framework. When the conditions are set, the user is notified only about the modifications of the documents that satisfy the conditions. We call these the modified nodes “interesting nodes.” If other users modify the global application state, only the data related to the interesting nodes is forwarded to this user. The conditions are declared as a set of rules for each client connection to decide when to move or copy data. Agents for event distribution (described below) use these conditions to acquire the objects for the application.

Event Distribution

Once the collaborative session is set up, the user actions are distributed as commands to the remote clients. The simplest way to transmit the commands between the clients and the servers is by using Java serialization. We employ this method in case of the applet implementations. However, the J2ME CLDC (Java 2 Micro Edition, Connected, Limited Device Configuration) does not support serialization and another method must be used for the Palm Pilot implementation. Our present version converts each command into a text string that is sent to the peer where it is converted back into a command.

Adapting Data to Device via Transformers

As the objects are to be visualized on very different devices, with very different computing and communication capabilities, some transformation of content must be performed. If some clients are visualizing in three-dimensional and others in two-dimensional, the three-dimensional coordinates have to be translated into two-dimensional and vice versa. The simple solution of just discarding or adding a z-coordinate would not work since, for example, the rotations around the x- or y-axis would be very complicated to handle.

A solution could be to keep data the same on all devices and transform locally if needed. This may sound like a feasible solution, but is not the best idea. Naturally, a three-dimensional workstation with all support from Java3D for transformation can fairly easy transform coordinates, but if the z-coordinate is not present in the first place, one cannot do much about it. The original data must be in a three-dimensional form. Transforming coordinates on the two-dimensional clients might not be a good idea either, because the clients running two-dimensional most likely do it for performance reasons and cannot handle additional burden of the transformation. Another problem occurs if Palm Pilots are involved. Even if visualization is only done in two-dimensional, the J2ME CLDC does not support floating-point numbers, so the coordinates must be converted to integer numbers, thus making it impossible to keep the same data format on all the clients. The result is that the distributed repositories are matched to the platform capabilities and user interests.

The command events must be transformed to a platform-specific format before being delivered remotely. The command transformation must be performed in such a

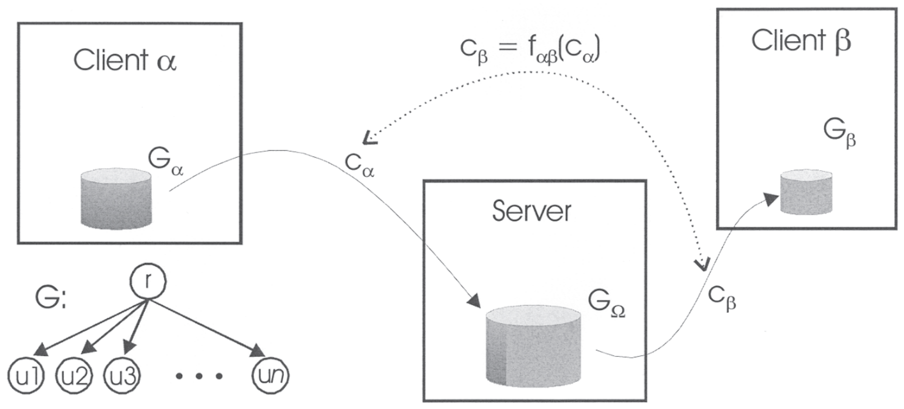

Figure 4. Command Transformations. The structure of the repository graph G with nodes u_i is shown in the lower left corner.

way to maintain the repositories in synchrony. The process is illustrated in Figure 4. Let us assume that the user interaction at client α results in a command c_α. A command consists of a sequence of primitive operations on the repository graph G (see the hierarchy of commands in Figure 2). These are: Op_1, create a vertex u_i; Op_2, delete a vertex u_j; and Op_3, modify vertex property. Using a transformation $f_{\alpha\beta}$, the command c_α gets transformed to c_β for delivery to site β. The transformation takes place in two stages. In the first stage, the transformer decides on whether a particular primitive operation in the command applies at the remote site. The result of the decision is binary: true or false. For example, if c_α is as follows:

$$c_\alpha = Op_1 \circ Op_1 \circ Op_3 \circ Op_1 \circ Op_2 \circ Op_3,$$

then c_β may be as follows:

$$c^1{}_\beta = f^1{}_{\alpha\beta}(c_\alpha) = Op_1 \circ Op_3 \circ Op_1 \circ Op_3.$$

The agents for event distribution perform this task. In the second stage, the properties associated with the operations of the type Op_3 must be converted to match the remote domain. The final result will be:

$$c_\beta = f^2{}_{\alpha\beta}(c^1{}_\beta) = Op_1 \circ Op^\beta{}_3 \circ Op_1 \circ Op^\beta{}_3.$$

As the server does not know anything about application-specific details, the application itself has to provide the necessary transformers in the form of agents and property converters. This is done when a client connects to the server for the first time. As part of the log in command sent to the server, the client can specify a URL where the server can load the necessary agents and converters if any. The server loads and instantiates the agents and converters to use them when communicating with the client. The server calls a method on the commands called convert with the appropriate converter as argument. The *convert* method of the commands then calls the necessary methods of the converter, depending on the kind of command. More details on transformations are available in Marsic et al. [17].

Agents for Event Distribution

As our system is targeted for wireless environments where network resources are sparse, we try to minimize the network traffic by using agents for event distribution. The agents filter the events sent to the user according to the users requirements, thereby saving network resources. Each user decides on what subset of the data he or she is interested in receiving notifications about, and sets up an agent at the server to decide which updates are sent back to the user. In order to respond to the very dynamic nature of wireless networks, our agents support priorities for event distribution. When configuring the agent, the user can specify the priority of different subsets of the data he or she is interested in, so updates about the least important data is only sent if network conditions permit, and the most important updates are sent even at very bad network conditions, using three levels of priority. This allows the event distribution to degrade gracefully when network conditions are getting worse.

An example of an application with users interested in different subsets of the full data set is a firefighter application for large operations, such as forest fires, in which we have clients at fire company (one fire engine), fire battalion, and fire division level. Each fire captain (leader of a fire company) will want to see the location of his fire fighters, neighbor fire companies, and known fire front close to the position of his company. The battalion chief will be interested in seeing only each of the companies as a unit and not the individual fire fighters within. Also, the chief might want to see his neighbor battalions, the entire fire front within the range of 10 miles, and all known evacuation vehicles within the range of 30 miles.

These hierarchies in the real world also impose a hierarchy in the collaborative applications. Although the data is the same, the participants should be able to specify the subset of the data the respective participant is interested in. Moreover, this subset is dynamic, since the conditions can change. For example, the company commander can become interested in specific squads if their mission is critical for the company's mission.

The agents are specified upon log in of the client, where the initial condition and the URL of the agent to use is sent to the server. The server loads the agent and initializes it with the initial condition, where the agent will then filter the events sent to the client. The user can change the conditions dynamically in response to changing requirements. Each agent is application-specific and knows the structure of the document that is shared by the clients and the server. In this way, the server remains completely independent on the application semantics. The server keeps an instance of the agent for each client. The agents are simply Java classes that have to implement a specific interface in order to collaborate with our system.

In the current version of the middleware implemented, the event-distribution agents use JESS expert system [5] and are able to evaluate at run-time the conditions sent by the clients.

The event notification service handles the distribution of events sent to the peers of the node. It is not expected that the event notification component will be replaced, but this is allowed if necessary.

When defining the core mechanism of the event notification service, some issues had to be taken into account. The requirements stated that the service should be replaceable, but we found that, rather than having to replace the whole service, we would, in most cases, only need to replace the decision-making part of the service. The decision-making part of the service can be seen as two separate tasks. (1) Deciding if an event should be forwarded based on the configuration of the system and the nodes role. For example, in a typical client-server system, a server node will forward all events, whereas a client node will only forward events originating from itself. We call the implementation of this task the event notification policy. (2) Deciding on which events to forward based on rules set up by the users; for example, a specific user is only interested in notifications about updates on some specific documents or even elements. We define this as the distribution rules.

The base for the first task is, in most cases, static, unless a node changes role, whereas the second often will be dynamic, as the user can change the rules at runtime. The second task can also involve decisions based on dynamic input such as changes in available resources. As a result, we defined two components to do the two tasks of the decision-making implementing the two interfaces: ReplicationPolicy and DataDistributionAgent. Using the components defined by these interfaces, the event notification service decides to which of the peers a given event is sent to and passes the message to the appropriate connections. A client node in a client-server configuration will typically have a very simple DataDistributionAgent and a very simple policy, because it is normally only connected to one peer—the server, whereas the server, on the other hand, will have a more complex DataDistributionAgent and more complex policies, since it simultaneously deals with multiple clients.

The event notification service keeps track of the active connections, getting notified by the controller when connections are added or removed, and by the connections when their state is changed. It can also be attached to a resource manager, if present, to get notifications about changes to available or allocated resources. This can be used by the DataDistributionAgent to distribute the events depending on available resources.

Since each message carries data of a single data type, we can assign priority to the message given the user defined rules for this data type. There are well-known methods in data networks for allocating different service qualities to messages with different priorities, which is called priority-based scheduling [14]. Figure 5 shows the architecture of the data distribution mechanism at the producer node. The scheduler works in a round-robin manner, but may have different strategies for sending the queued messages, called queuing discipline. It may send all high-priority messages first, or it may assign higher probability of sending to the high-priority messages, but the low-priority messages still get nonzero probability of being sent. The queue buffers are finite, and if a buffer overflows, the subsequent messages are dropped.

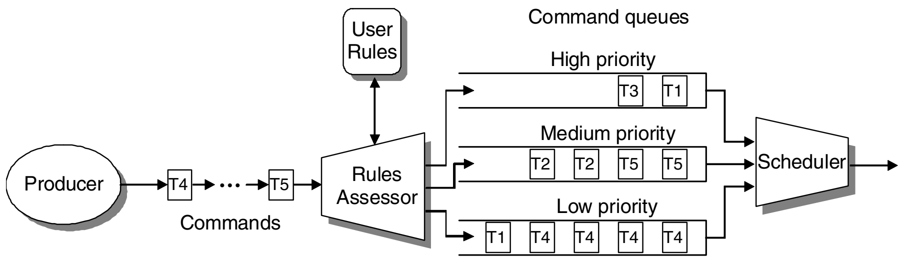

Figure 5. Priority Scheduling of Commands. User Rules and the Rules Assessor make the DataDistributionAgent.

Sample Applications

FOUR COMPLEX EXAMPLE APPLICATIONS have been developed using the infrastructure provided by Manifold and DISCIPLE (see Figure 6): a two-dimensional graphics editor (Flatscape), a three-dimensional virtual world (cWorld), a two-dimensional graphics editor for Palm Pilots (Palmscape), and a game called Slow Tetris used in human–computer interaction experiments with heterogeneous views. Both Flatscape and cWorld is built on a full version of DISCIPLE and Manifold, and run on desktop workstations, whereas Palmscape is using only the core classes of DISCIPLE and Manifold.

The Slow Tetris application is an extension to Flatscape and is therefore using the full DISCIPLE and Manifold code both for PCs and handheld devices.

Flatscape

Flatscape is an expandable two-dimensional graphical editor with the typical functionality of such editors. New applications can be built upon it, and, among other things, it has been used to build online virtual collaborative laboratories for biology teaching [26]. Flatscape is developed using Java2D. Two example applications of Flatscape are shown in Figure 7.

The first application was built using the earlier middleware (Uform-based). The telepointers in Flatscape are two-dimensional arrows pointing away from the center of the document (room, map, etc.) toward the part of the document currently viewed by the user. This is different from the typical telepointer, such as that described in Roseman and Greenberg [23], which shows only position and not orientation. In Flatscape implementation, the telepointer owner, rather than the recipient, has control over its visibility. The telepointers, thus, can be considered as primitive avatars.

The second application was built using the current middleware (XML-based) targeted for collaboration on a situation map. In this case, Flatscape uses a document structure defined in a XML Schema including elements such as layers for time and symbol grouping, besides the symbols themselves. Flatscape has been implemented to be used on both traditional PCs and wearable computers, and even a version for

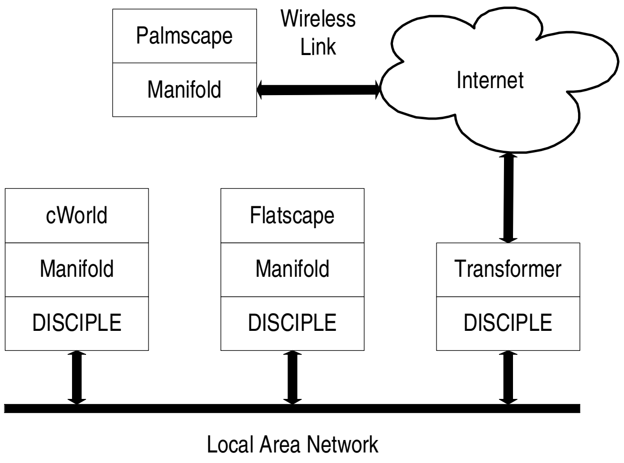

Figure 6. The System Architecture and the Example Applications: Palmscape, Flatscape, and cWorld.

PDAs has been made. All versions supports operations such as view a map, zoom in and out, place, move, rotate and delete objects, freehand drawing, object property modification, and so on.

cWorld

cWorld application (Figure 8) enables synchronous, multiuser building of collaborative virtual environments (CVEs) using Java3D. cWorld does not require special hardware and can be operated using the keyboard and a mouse, although it also supports the use of the Magellan SPACE Mouse. This device provides the six-degrees-of-freedom movement used in navigating three-dimensional spaces.

cWorld allows the users to directly manipulate objects in the CVE, such as rotate or move, by providing a set of manipulation tools. Each user has a unique three-dimensional telepointer (Figure 8, left). These devices function as primitive avatars and appear at the discretion of the user. The telepointer is drawn at the position and orientation of the user's line of sight.

Figure 8, shows an example application, where, in the three-dimensional interface, the user is looking at a three-dimensional model of an office. The user can create an empty office, navigate, place objects (pieces of furniture), select objects, move objects, and show other users his or her position and orientation using the telepointer. The tools the system provides to perform these tasks are implemented in the cWorld toolbar. Only one of the tools is active at a given time. A white background of the active button represents an active tool. For navigating in the room and to move the object once placed, the user will is using the Magellan three-dimensional mouse device.

In order to place objects (pieces of furniture) in the scene (office), a user follows two steps: first, the user selects from the toolbar the object to be added and then clicks on the three-dimensional scene to place the object.

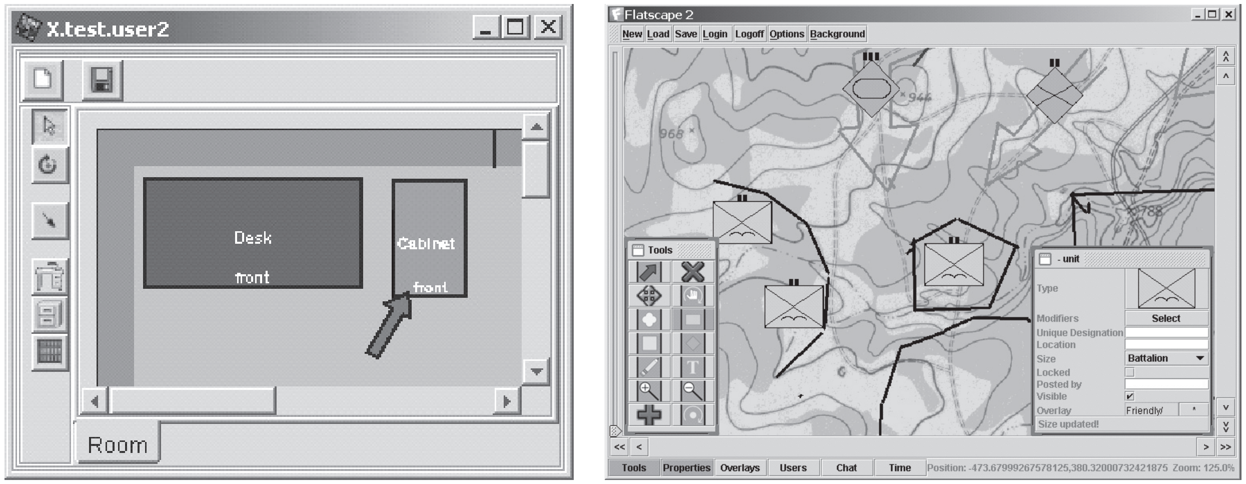

Figure 7. Two Sample Applications Built Using Flatscape. Left: A room floor plan showing a two-dimensional telepointer. Right: A collaborative map.

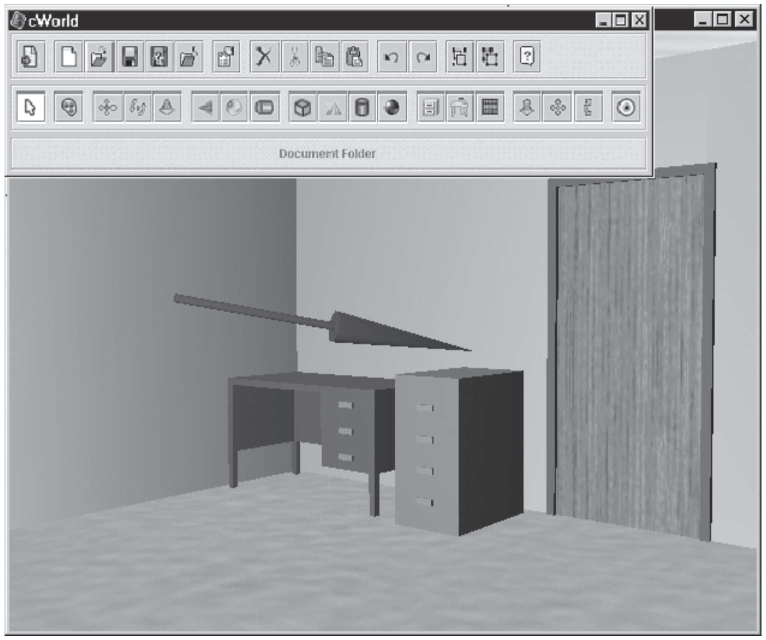

Figure 8. Left: CVE Built Using cWorld (corresponding to the sample application shown in Figure 7, left). The room snapshot also shows a three-dimensional telepointer. Right: Logitech Magellan SPACE mouse used as input pointing device for cWorld.

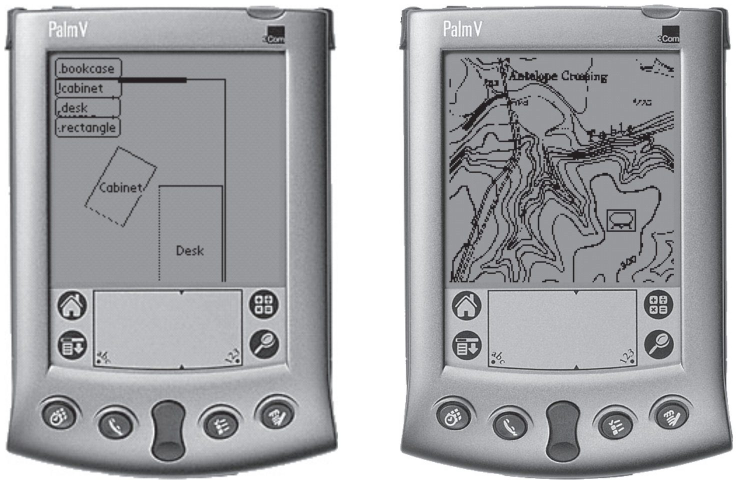

Figure 9. Left: The Sample Application Shown in Figure 7, Left, and Figure 8 Running in Palmscape on a Palm V Handheld Device. The room snapshot shows the menu for adding objects. Right: The collaborative map shown in Figure 7, right, running in Palmscape on a Palm V handheld device.

Palmscape

Palmscape application (Figure 9) is developed using J2ME CLDC 1.0 [27]. It is a simpler version of the Flatscape editor, built using the basic Manifold interfaces, but still with most of the same basic functionality (e.g., users can create, delete, move, and rotate objects) and user input is via the Palm stylus and buttons. Unlike Flatscape, it does not support multiple documents. In the present version, telepointers are not implemented.

Object Example

Figure 10 shows different views of the same piece of furniture: a file cabinet shown as three-dimensional on a PC screen (cWorld), as two-dimensional on a PC screen (Flatscape), and as two-dimensional on a Palm V screen (Palmscape).

Slow Tetris Game

In a project studying user behavior when collaborating in heterogeneous environments [28], we developed a block manipulation game using the DISCIPLE middleware. The game was developed in two versions; a two-dimensional version to run on pocket PCs and a three-dimensional version to run on desktop PCs as shown in Figure 11.

We call the game Slow Tetris since it is somewhat similar to the popular two-dimensional Tetris game. The object of the task is to build a wall from a series of building blocks that are supplied to the collaborators. Figure 11 illustrates an example set of block shapes that can be built into a wall in the three- and two-dimensional display,

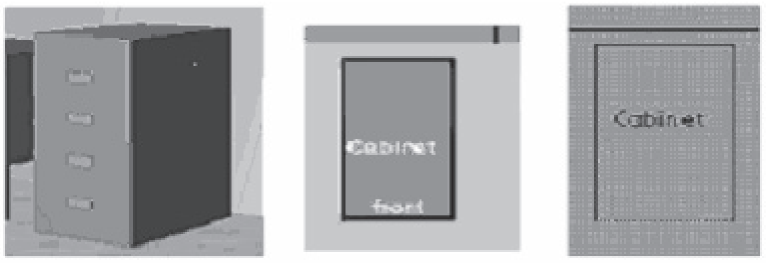

Figure 10. Different Views of the Same Piece of Furniture (file cabinet) in cWorld (left), Flatscape (middle), and Palmscape (right).

respectively. The first block is fixed and the subsequent blocks are in the order they are to be placed in the wall. The subsequent blocks have to be rotated into the correct orientation so that they fit into the wall being built.

Evaluation

THE DEVELOPMENT OF THE SAMPLE APPLICATIONS was an evaluation in itself, assessing the feasibility of using the Manifold framework to develop applications for heterogeneous environments. To evaluate the DISCIPLE and Manifold, we selected two applications, Flatscape and Palmscape, and used them to measure the performance of the system in a map-based collaboration scenario. To allow users to communicate during collaboration, we augmented them with a basic chat tool (provided by DISCIPLE), with the options of setting up filters for exclusion of messages from specific users.

Reusability

Table 1 shows the statistics of the class reuse for the communication server, Flatscape, and Palmscape, as total number of used classes, number of classes belonging to each application (own), number of classes reused from Manifold, number of classes reused from DISCIPLE, and the percent of reused classes (from the Manifold and DISCIPLE classes together). The percentage is computed as the ratio between the number of reused classes and the total number of classes.

Table 2 shows the statistics of code reuse in terms of number of lines of code in different Java packages. As the tables show, the framework provides a high degree of code reuse on very different platforms. This is conformed in our current experience with developing more applications based on this framework.

Reliability

As this system is not a production system, we have not measured the up-time of it, but have focused on the system to be reliable in the sense of being able to maintain consis-

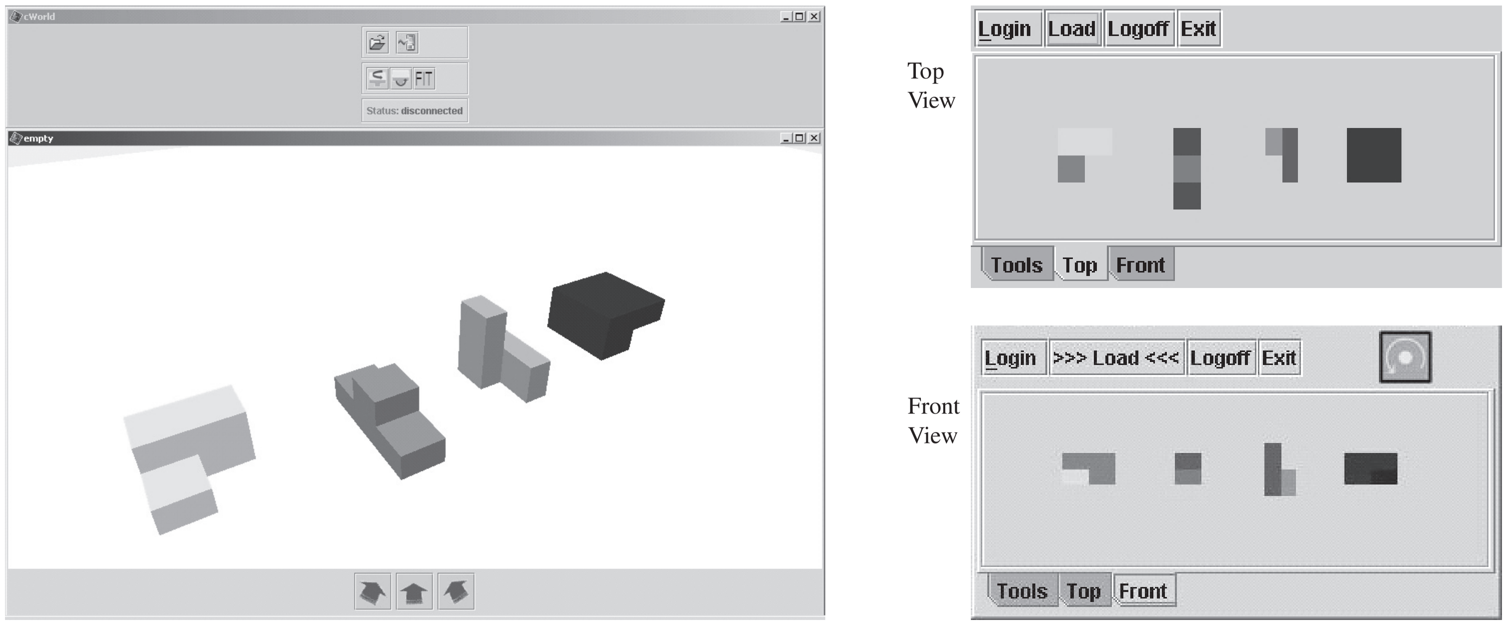

Figure 11. Screenshots of Two Embodiments of the Slow Tetris Game in Flatscape. Left: The three-dimensional version. Right: The two-dimensional version visualizing the same data objects as shown in the three-dimensional version.

Table 1. Statistics of Class Reuse for Each Application.

	Number of classes				
Application	Total	Own	Manifold (M)	DISCIPLE (D)	Percent reuse ($M + D$)/total
Server	50	3	0	47	94
cWorld	118	58	13	47	51
Flatscape	91	31	13	47	66
Palmscape	48	21	5	22	56

Table 2. Statistics of Code Reuse in Terms of Lines of Code for Each Application.

	Number of classes				
Application	Total	Own	Manifold (M)	DISCIPLE (D)	Percent reuse ($M + D$)/total
Server	5,715	660	0	5,379	89
cWorld	13,599	7,551	669	5,379	44
Flatscape	9,409	3,361	669	5,379	64
Palmscape	4,413	1,216	336	2,518	70

tency of the data. While the clients are connected to the server it is easy to ensure consistency, because the server is replicating all events in the order they are received and all the clients therefore receives the events in that order; that is, global ordering. When clients disconnect and thereafter connect again, the problem becomes more complicated. Ionescu et al. [10] further developed the original state merging algorithm and Marsic et al. [17] provided a formal proof for the algorithm correctness in the case of maintaining state consistency across heterogeneous collaborative applications.

Scalability

We evaluated the performance of the current middleware version using two criteria: (1) the event processing time versus number of clients and (2) the number of data objects.

Event Processing Time

We measured the event processing time versus the number of clients with and without the DataDistributionAgent. The server application generates a number of events that are forwarded to the connected clients.

Testing with the DataDistributionAgent clients must subscribe to events at the server to receive events. Initially, clients subscribe with the rules that they want the server to filter events with. Once the server receives the rules, the rules are inserted into a rule-base. The rule-base contains all subscription rules and is maintained by the DataDistributionAgent. When the server receives an event, the DataDistributionAgent runs the rule-base, matching rules against the event. Rules that match fire and the DataDistributionAgent collect the fired rules and return a list of clients that are interested in the event. The rules used in the tests are simple. The set of rules the server receives from each client consists of two rules. The first rule is a subscription to a common document used by all collaborating clients. The rule fires if an event is of type org.w3c.dom.Document with the name "#document." The second rule subscribes to the events of type org.w3c.dom.Element named "tag." The events generated by the server are all alike, an org.w3c.dom.Element named "tag." The rules are not complex. They have a depth of one, meaning that there is only one condition that needs to be satisfied before the rule fires.

Testing without the DataDistributionAgent, the server bypasses the filtering mechanism and sends all events to all connected clients without considering if the clients are interested in the event.

Four cases were evaluated: running clients and the server on one and the same machine (Localhost test) with and without DataDistributionAgent, and running clients on three separate machines and the server on a fourth machine (Distributed test) with and without DataDistributionAgent. When testing with DataDistributionAgent, we considered the worst-case scenario, where all rules fire at every incoming event, thus all events were sent to all clients.

The time measured was the time from when an event was received at the server until it was sent to all relevant clients. We created 100 events and measured the time the server took to process all 100 events. Comparing the two, we evaluated the overhead introduced by the DataDistributionAgent. The overhead should not outweigh the savings in network resources.

The evaluation was performed on Windows desktops. The server was run on a DELL 2.4 GHz Intel XEON with 1,024 MB RAM running Windows XP Pro. When running the distributed test, clients ran on a DELL 2.2 GHz Intel XEON with 1,024 MB RAM running Windows XP Pro, a IBM 600 MHz Intel Pentium III with 512 MB RAM running Windows XP Pro, and a DELL 1.4 GHz Pentium III with 1,024 MB RAM running Windows 2000 Pro.

We found a very close to linear relationship between the event processing time and the number of clients. This is shown in Figure 12.

The event processing time for one event testing with 500 clients is 450 ms with rules and 40 ms without rules for the Distributed test and 800 ms with rules and 400 μs without rules for the Localhost test. The reasons for the higher processing time when using rules are (1) the evaluation of rules are based on comparison of strings and (2) this is a worst-case scenario; all rules constantly fire when a new event is evaluated.

The higher event processing times when running the Localhost test with rules is attributed to the server and all clients are run on and maintained by the same machine. We find these event processing times acceptable, especially since there is room for optimization in the current implementation.

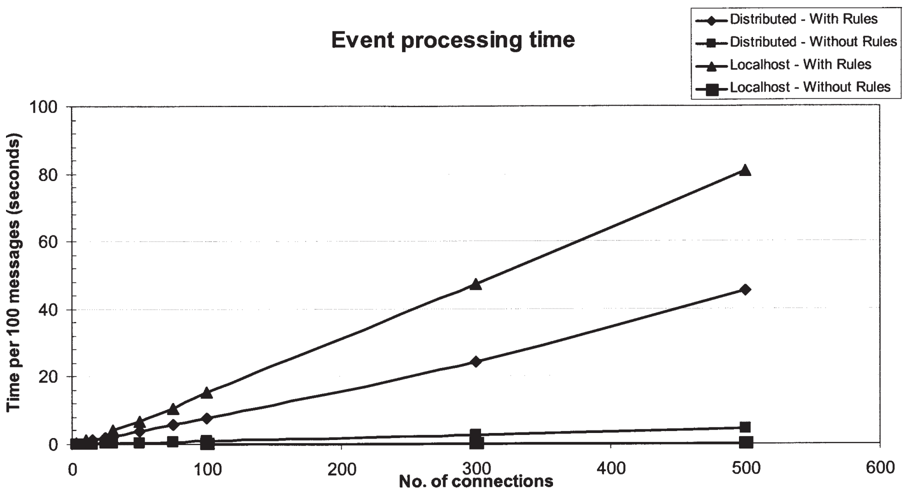

Figure 12. Event Processing Time by Number of Clients.

Number of Data Objects

The number of data objects possible to store in the BasicRepository depends on the implementation of the java.util.Hashtable (storing the documents), on the implementation of the org.w3c.dom.Node (key element of the DOM structure), and on the available memory of the server. As both the Hashtable and Node[1] are implemented rather robustly and can handle sufficient large amounts of data objects, the main constraint comes from the amount of memory available. We tried on a PC with 512 MB of physical memory to create 1 million objects. This went fine until around 500,000 elements when the PC ran out of memory and therefore started to cache on the disk. The server continued to operate correctly, but was very slow because of the disk caching. This simple test indicates that, with a sufficient amount of memory, our middleware can handle large data structures. If the need is for even larger structures, the best solution will probably be to implement a Repository, which serves as a front end of commercial database providing XML support.

Conclusions

We have designed and built a system that targets many of the problems with collaboration in heterogeneous environments having slow or unreliable connections. The system is based on a distributed repository, supported by a client-server infrastructure, and a framework for building heterogeneous applications. We presented example applications. The experimental evaluation demonstrates that the framework scales well to platform computing capabilities, provides significant code reuse, and offers good performance, mainly limited by the network latency, which we cannot do much about if we want to have connectivity on the go.

Our future work gives us two directions to go in data adaptation at the application level and communication infrastructure level. Using the first implementation of the communication infrastructure, we are in the process of developing a shared calendar, a text editor, and a resource management application, which we plan to use internally in our group. We will monitor the use of the system as a field trial to obtain information about the usability of the system. We are also investigating the use of more complex transformers between the heterogeneous applications. Distributing and synchronizing the repositories between multiple servers is another current area of research. Other areas for investigation include how a client can be implemented on future cell phones and other small devices, such as pagers.

Our second design of the communication infrastructure shows that it is possible to develop a flexible middleware, where almost all components are replaceable, and still obtain a reasonable performance and robustness (depending on the implementation of the noncore components). It opens the path for research in implementing different types of repositories and dynamic adaptation of the communication between client and server as well as between servers for changing connection quality. This gives us a good foundation for the continuation of our research in middleware for real-time collaboration. With the increasing interest in the use of small connected devices, we also see more interest in peer-to-peer configurations, both for spontaneously connected devices, and for webs of peer-to-peer connected devices. At the same time, quality of service (QoS) is expected to become more of an issue with the users beginning to pay for the services—a user paying for a service will expect a certain level of QoS, or else wants to pay less for the service.

Our main focuses for the future are therefore on peer-to-peer configurations and resource management. We want to investigate state replication and synchronization, as well as event distribution and routing in peer-to-peer configuration, which we hope will lead to a default peer-to-peer implementation of the DISCIPLE middleware. We also want to develop a resource management component based on research in passive and active bandwidth measurement currently being performed in our group. The resource management component should act as a resource broker, allocating resources to its clients, and, at the same time, monitoring these resources and notifying the clients upon changes.

Acknowledgments: Mihail Ionescu and Jorgen Nielsen from CAIP Center contributed to the DISCIPLE middleware with research and software development. The research presented is supported by NSF Contract No. ANI-01-23910, U.S. Army CECOM Contract No. DAAB07-02-C-P301, and by the Center for Advanced Information Processing (CAIP) and its corporate affiliates.

Note

We are using the Crimson implementation packaged in the JAXP 1.1 release, available at java.sun.com/xml.

References

1. Carzaniga, A.; Rosenblum, D.S.; and Wolf, A.L. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems, 19,* 3 (2001), 332–383.

2. Dewan, P., and Sharma, A. An experiment in interoperating heterogeneous collaborative systems. In S. Bødker, M. Kyng, and K. Schmidt (eds.), *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99).* Copenhagen: Kluwer Academic Publishers, 1999, pp. 371–390.

3. Duran-Limon, H.A.; Blair, G.S.; Friday, A.; Grace, P.; Samartzidis, G.; Sivaharan, T.; and Wu, M. Context-aware middleware for pervasive and ad hoc environments. European Commission, Information Society Technologies (IST) Programme RTD Research Project IST-2000-26031, Lisbon, Portugal, 2003 (available at cortex.di.fc.ul.pt/).

4. Engelbart, D.C. Authorship provisions in AUGMENT. Paper presented at the Intellectual Leberage: The Driving Technologies Compcon 84 (Twenty-Eighth IEEE Computer Society International Conference). Los Alamitos, CA: IEEE Computer Society Press, 1984.

5. Friedman-Hill, E. JESS: The Java expert system shell. Sandia National Laboratories, Livermore, CA, 2003 (available at herzberg1.ca.sandia.gov/jess/).

6. Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. *Design Patterns.* Reading, MA: Addison-Wesley, 1995.

7. Higgins, M.; Lucas, P.; and Senn, J. VisageWeb: Visualizing WWW data in Visage. Paper presented at the Proceedings of the IEEE Symposium on Information Visualization (InfoVis'99), San Francisco, October 1999.

8. Hill, R.D.; Brinck, T.; Rohall, S.L.; Patterson, J.F.; and Wilner, W. The rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction, 1,* 2 (1994), 81–125.

9. IBM Alphaworks. XML TreeDIFF. IBM, San Jose, CA, 1998 (available at www.alphaworks.ibm.com/tech/xmltreediff).

10. Ionescu, M.; Krebs, A. M.; and Marsic, I. Dynamic content and offline collaboration in synchronous groupware. Paper presented at the Proceedings of the Collaborative Technologies Symposium (CTS 2002), San Antonio, TX, 2002.

11. Karsenty, A.; Tronche, C.; and Beaudouin-Lafon, M. GroupDesign: Shared editing in a heterogeneous environment. *USENIX Computing Systems, 6,* 2 (1993), 167–195.

12. Keshav, S. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network.* Reading, MA: Addison-Wesley, 1997.

13. Kortuem, G., and Schneider, J. An application platform for mobile ad-hoc networks. Paper presented at the 2001 Ubiquitous Computing Conference, Workshop on Application Models and Programming Tools for Ubiquitous Computing (UBICOMP 2001). Atlanta, GA, 2001.

14. Kraut, R.; Miller, M.D.; and Siegel, J. Collaboration in performance of physical tasks: Effects on outcomes and communication. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'96).* New York: ACM Press, 1996, pp. 57–66.

15. Krebs, A.; Ionescu, M.; Dorohonceanu, B.; and Marsic, I. The DISCIPLE system for collaboration over the heterogeneous web. In R.H. Sprague, Jr. (ed.), *Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences.* Los Alamitos, CA: IEEE Computer Society Press, 2003 (available at csdl.computer.org/comp/proceedings/hicss/2003/1874/00/1874toc.htm).

16. Litiu, R., and Prakash, A. Developing adaptive groupware applications using a mobile component framework. In W. Kellogg and S. Whittaker (eds.), *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW 2000).* New York: ACM Press, 2000, pp. 107–116.

17. Marsic, I.; Sun, X.; Correa, C.D.; and Liu, T. Maintaining state consistency across heterogeneous collaborative applications. CAIP-TR-264, Center for Advanced Information Processing (CAIP), Rutgers University, Piscataway, NJ, 2002 (available at www.caip.rutgers.edu/disciple/Publications/CAIP-TR-264.pdf).

18. Mascolo, C.; Capra, L.; Zachariadis, S.; and Emmerich, *W.* XMIDDLE: A data-sharing middleware for mobile computing. In R. Prasad and T.K. Madsen (eds.), *Wireless Personal Communications.* Dordrecht: Kluwer, 2002, pp. 77–103.

19. Maya Design Group, Inc. Visage link. Pittsburgh, 2003 (available at www.maya.com/visage/base/technical.html).

20. Meier, R., and Cahill, V. STEAM: Event-based middleware for wireless ad hoc networks. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*. Vienna, Austria, 2002 (available at csdl.computer.org).

21. Pietzuch, P., and Bacon, J. Hermes: A distributed event-based middleware architecture. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*. Vienna, Austria, 2002 (available at csdl.computer.org).

22. Preguiça, N.; Legatheaux Martins, J.; Domingos, H.; and Duarte, S. Data management support for asynchronous groupware. In W. Kellogg and S. Whittaker (eds.), *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'00).* New York: ACM Press, 2000, pp. 69–78.

23. Roseman, M., and Greenberg, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction, 3,* 1 (1996), 66–106.

24. Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium (W3C), Cambridge, MA, 2000 (available at www.w3c.org/TR/SOAP).

25. Stefik, M.; Bobrow, D.G.; Lanning, S.; and Tatar, D.G. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Information Systems, 5,* 2 (1987), 147–167.

26. Subramanian, R., and Marsic, I. ViBE: Virtual biology experiments. In V.Y. Shen, N. Saito, K.T. Yung, M.R. Lyu, and M.E. Zurko (eds.), *Proceedings of the Tenth International World Wide Web Conference (WWW10).* New York: ACM Press, 2001, pp. 316–325.

27. Sun Microsystems, Inc. Java J2ME connected limited device configuration (CLDC). JavaSoft. Santa Clara, CA, 2003 (available at www.javasoft.com/products/cldc/).

28. Velez, M.; Tremaine, M.; Sarcevic, A.; Dorohonceanu, B.; Krebs, A.; and Marsic, I. "Who's in charge here?" Communicating across unequal computer platforms. *ACM Transactions on Computer-Human Interaction,* forthcoming.

29. Whitehead, J., and Goland, Y.Y. WebDAV—A network protocol for remote collaborative authoring on the Web. Paper presented at the European Computer Supported Cooperative Work (ECSCW'99) Conference, Copenhagen, Denmark, 1999.

30. XML Linking Language (XLink), Version 1.0. W3C Recommendation. World Wide Web Consortium (W3C), Cambridge, MA, 2001 (available at www.w3c.org/TR/xlink/).

31. XML Path Language (XPath), Version 1.0. W3C Recommendation. World Wide Web Consortium (W3C), Cambridge, MA, 1999 (available at www.w3c.org/TR/xpath/).