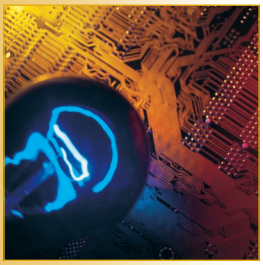

Adaptive Collaboration for Wired and Wireless Platforms

A data-centric architecture for collaboration environments uses XML to adapt shared data dynamically between devices with widely disparate capabilities.

Ivan Marsic
Rutgers University

Expanding the Internet's reach with wireless links and mobile terminals establishes an infrastructure that permits not only individual roaming but also, potentially, interactive collaboration in a more complex workspace. The classic example is an expert using a 3D CAD model on a workstation to collaborate with someone in the field using a handheld device. The possibilities for collaboration will become more elaborate with advances in visualization technologies for small portable devices (for example, see the MiniGL 3D graphics library from Digital Sandbox, http://www.dsbox.com/minigl.html, and the Pocket Cortona library from ParallelGraphics, http://www.parallelgraphics.com/products/cortonace/).

The obstacles to such collaboration hinge on significant disparities between the platforms in network capabilities, interaction modalities, computing power, and storage. One user might have broadband wired service while another has mere voice-bandwidth wireless, making it difficult to meet any quality-of-service (QoS) application requirements. Client equipment might range from high-end workstations with multimodal interfaces and elaborate graphics to cell phones, palmtops, or wearable computers with small 2D monochrome displays.

To address these issues, the Disciple[1] and Manifold[2] frameworks, developed at Rutgers Center for Advanced Information Processing (CAIP), take a data-centric approach to the development of collaborative multimedia applications. (Disciple is an abbreviation for Distributed System for Collaborative Information Processing and Learning; for related work, see the sidebar.) This approach represents data and modifies it in a generic way via a standard representation medium and a standard communications protocol. Each user can obtain a subset of the shared data that can be visualized differently for different users (polymorphic views). The amount of data presented and the visualization technique reflect a user's interests

 1089-7801/01/$10.00 ©2001 IEEE

Related Work in Collaboration Across Dissimilar Devices

Many collaborative systems and toolkits address application and data sharing across dissimilar terminals.

Shared Displays

Garfinkel et al.[1] developed SharedX, which adapts shared displays to various devices by degrading image quality, where necessary, to match the hardware capabilities. The authors assumed, however, that users would have the same replicated application and just apply device-specific color-space reduction.

Rendezvous,[2] GroupKit,[3] and several other groupware toolkits employed model-view separation that would let developers create distributed data models and drive different views, but no implementations with distributed models have been reported. In some cases (like Rendezvous), a centralized groupware architecture simplifies development and deployment.

Groupware Interoperability

True heterogeneity requires also interoperability between the existing groupware systems. Dewan and Sharma[4] focused on interoperating group systems with different policies (concurrency control, coupling) and architectures (centralized versus replicated), but did not consider how computing platform heterogeneity affects semantic consistency.

The Mash project[5] introduced application-level proxies for adaptation between heterogeneous clients communicating over heterogeneous networks. The authors applied their architecture to PDA-based Web browsing and to collaboration between desktop and PDA clients. In their system, a PDA essentially acts as a remote display for a proxy that maintains shared application state, making support difficult for disconnected client operations.

XML-Based Adaptation

Several efforts are under way to realize XML's potential for information exchange on heterogeneous devices. The Wireless Application Protocol (WAP) Forum has standardized the Wireless Markup Language (WML),[6] an XML language optimized for specifying presentation and user interaction on limited-capability terminals such as cell phones. Companies like Nokia already offer WAP-compliant devices, but the WAP architecture does not utilize XML/XSL separation at this time.

To our best knowledge, there is currently no published research that uses XML/XSL separation — as Disciple does — for dynamic runtime adaptation of interactive application models and visualizations to heterogeneous devices.

References

1. D. Garfinkel, B.C. Welti, and T.W. Yip, "HP SharedX: A Tool for Real-Time Collaboration," *Hewlett-Packard J.*, vol. 45, no. 2, Apr. 1994, pp. 23-36.
2. J.F. Patterson et al., "Rendezvous: An Architecture for Synchronous Multi-user Applications," *Proc. ACM Conf. Computer-Supported Cooperative Work* (CSCW 90), ACM Press, New York, 1990, pp. 317-328.
3. M. Roseman and S. Greenberg, "Building Real-Time Groupware with GroupKit, a Groupware Toolkit," *ACM Trans. Computer-Human Interaction*, vol. 3, no. 1, Mar. 1996, pp. 66-106.
4. P. Dewan and A. Sharma, "An Experiment in Interoperating Heterogeneous Collaborative Systems," *Proc. 6th European Conf. CSCW* (ECSCW 99), Kluwer, Copenhagen, 1999, pp. 371-390.
5. A. Fox et al., "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Comm.* (Special Issue on Adapting to Network and Client Variability), vol. 5, no. 4, Aug. 1998.
6. *Wireless Application Protocol: Wireless Markup Language Specification*, v. 1.3, approved specification WAP-191-WML, WAP Forum, Feb. 2001; available for download at http://www.wapforum.org/what/technical.htm.

and, more important, a device's computing and communications capabilities.

This article begins by introducing a data-centric architecture that abstracts collaborative tasks as editing of data repositories, followed by descriptions of the role of XML in managing heterogeneity and intelligent software agents in discovering network and computing environment conditions. Finally, I present an instantiation of a furniture arrangement application used to test the approach.

A Data-Centric Groupware Architecture

The Disciple architecture uses structured data and defines a universal format for representing it in a way that gives developers finer-grained control over data adaptation. Specifically, Disciple uses XML for data representation and exchange[3] and XSL (Extensible Stylesheet Language)[4] for data presentation and cross-domain adaptation.

This data-centric method of application sharing is the key to application interoperability and adaptation. In essence, the task is supported by the data, and the user-data interaction is tailored to the device — rather than adapting the task to the device. Collaboration centers on a shared data set stored in repositories on distributed servers, and structured data represent hierarchically organized semantic objects. For example, a data set could represent an engineering drawing, text document, spreadsheet, or virtual world. For simplicity's sake we assume that the data are structured into a tree or, in a more general case, a directed acyclic graph. The tree structure might not be the most efficient data type for all applications, but settling on one data type simplifies groupware design considerably.

A collaborative application renders data objects and allows the user to interact with the renderings using textual, graphical, or multimedia commands. We designed an abstract data type, called a *uni-*

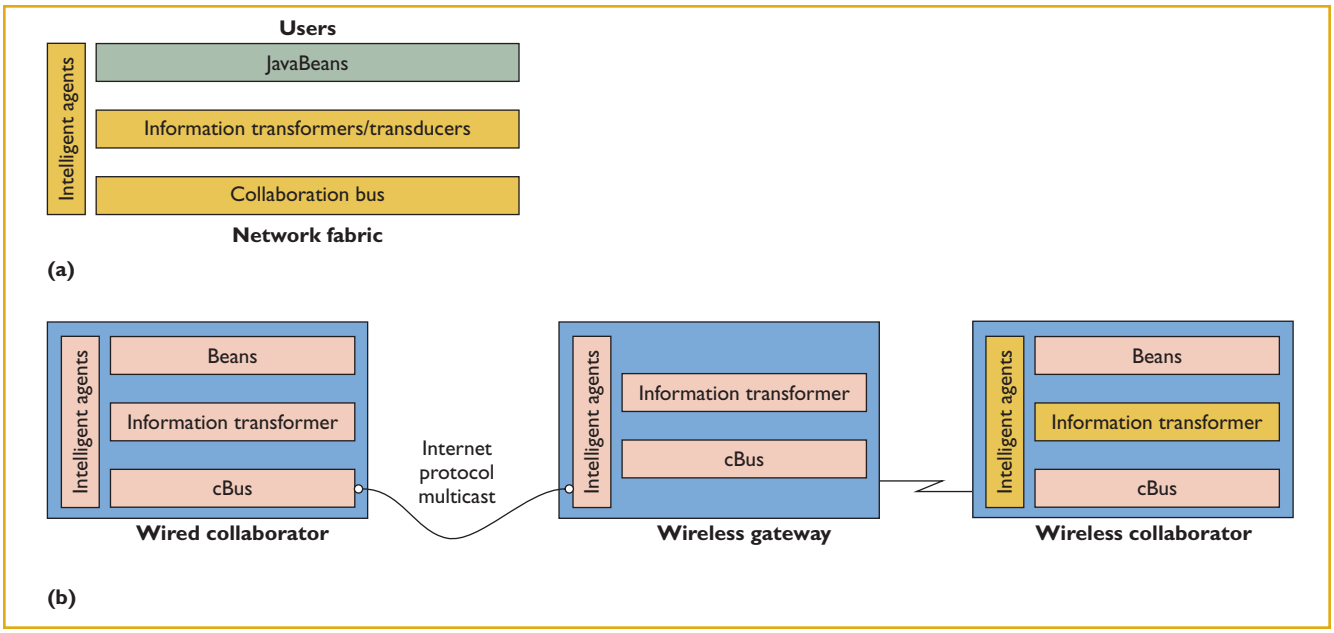

Figure 1. Architecture of the Disciple collaboration framework. (a) Framework components (yellow) do not include JavaBeans and applets, which are supplied by the application developer. (b) Distribution of components in a heterogeneous environment shows wireless clients constituting some or all of the yellow-colored components, depending on the access device's capabilities.

versal form, or UForm, to model the shared objects. This basic data unit consists of a globally unique identifier and a keyed list of properties, as also proposed for Visage.[5] Only three operations apply:

- create UForm,
- delete UForm, and
- modify UForm property.

The properties can be geometric (dimensions, color, or texture), but they can also be the IDs of other UForms, thus forming a hierarchical data structure.

Our data-centric approach follows the World Wide Web model of using a standard medium for representation (HTML/XML) and a communication protocol (HTTP) that does not prescribe client or server specifics. In our case, Disciple universally represents data as repositories of UForms, and the communication protocol permits accessing the UForms and performing the primitive operations on them. Both repositories and messages are represented in XML. Any client or server application that understands these representation and communication standards can communicate with any other such application. Developers or users can then tailor each application to a device type's available computing and network resources.

Disciple Framework

Figure 1 shows the architecture of the Disciple framework. Figure 1a shows the framework components. Figure 1b shows a simplified view of the architecture distributed in a heterogeneous environment, where wired clients belong to one class and wireless clients to another.

The servers use the same components as the clients except that server JavaBeans have some administrative functionality and, unlike client beans, do not support user interaction. The *wireless gateway*, an optional component, acts as a proxy that deals mainly with nonstructured data such as images and video. The *intelligent agents plane* ensures that its client can participate effectively in a heterogeneous collaboration session. It uses a knowledge base to track the client's interests, computing capabilities, and QoS requirements and translates them into a *client profile* that defines the type and level of information abstraction needed. The agents use the *information transformer module* based on the profile to adapt the incoming data for visualization and outgoing data for network transmission.

To transform data across diverse platform capabilities and user needs, the information transformer/transducer module maintains a suite of media-specific information abstraction algorithms. Information abstraction aims to reduce information content while maintaining semantics, ensuring fundamental object integrity even on bandwidth- and display-disadvantaged mobile wireless clients. Examples of information abstraction include level-of-detail variation, image-to-text,

text summarization, text-to-speech, and speech-to-text conversions.

Disciple's central component is the *Collaboration Bus*,[6] or cBus, which handles all common groupware processing that can be abstracted away from specific applications and built into the infrastructure. The cBus consists of named communication channels to which the applications can subscribe and publish information; it is the key to replicating state changes of the shared data repositories.

Servers maintain data repositories, which clients access as XML documents. A client joining a session accesses the repository contents it needs and stores them locally to allow offline work. While client and server are connected, the local and server repositories are maintained in synchrony. The system can, however, support synchronous and asynchronous collaboration simultaneously, which is important to associates working in different time zones or otherwise unable to participate in synchronous sessions. Unlike a proxy-based approach, the Disciple framework does not require continuous connectivity; users can continue their work even when the link temporarily goes down, for example, during a wireless connection. We might include proxies for data transformation, but the application logic (JavaBeans) always resides on a client rather than on its proxy.

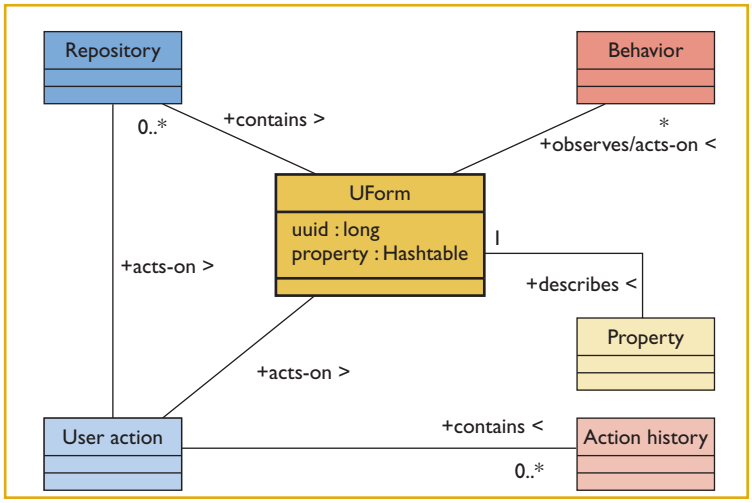

Figure 2. Conceptual model of a generalized editor, represented in Unified Modeling Language notation. Collaborative tasks are abstracted as editing of repositories of generic data units, UForms.

Sharing JavaBeans

Disciple is an *application framework*, that is, a semicomplete application that guides the creation of customized applications. End users (conference participants) complete their applications at runtime by selecting and importing task-specific JavaBeans into the Disciple workspace, a *shared container* that permits group sharing.

Each bean represents an application plug-in that enhances Disciple's generalized application-sharing abilities, much as Java applets enhance Web browser functionality. Collaborators select plug-ins based on the task at hand and import the selected bean into the workspace by drag-and-drop manipulation. The imported bean becomes a part of a multiuser application and all conferees can interact with it. Objects in the bean are not aware of distributed services or sharing. They interact with Disciple through the JavaBeans event model as with any other event source or listener, and Disciple distributes the data they generate.

Manifold Beans

Disciple handles heterogeneity primarily at the communication/networking level[7] but not at the application logic and interaction levels. We therefore created the Manifold framework[2] to facilitate the development of collaborative applications, which exist as JavaBeans, for heterogeneous computing environments. Manifold follows the Model-View-Controller design pattern and includes the base package with Java interfaces for an application's basic structure, along with default implementations for some of these interfaces to assist application developers.

Manifold has a multitier architecture that consists of vertical presentation, application or domain logic, and storage tiers. Manifold's *presentation* tier is virtually free of application logic and simply visualizes the domain data and accepts user input. The *domain* tier handles the semantics of data manipulation tasks and rules. Manifold's third tier manages the collaboration functionality, which is mainly provided by the Disciple framework but also partially resides in the Manifold bean.

The framework's data-centric focus permits abstracting user activity as data repository editing, a conceptual model shown in Figure 2. We can describe any task's application logic as a set of primitive UForm operation sequences. In addition to passive editing, a generalized editor could permit behavior objects triggered by UForm changes. Such behaviors include collision detection in 3D worlds, spreadsheet cells with formulas, or coordinated manipulation of several UForms. Other possible tasks include data searches and sorting.

The user does not, however, issue primitive UForm operations. To facilitate user interaction, the presentation tier visualizes the UForms and provides

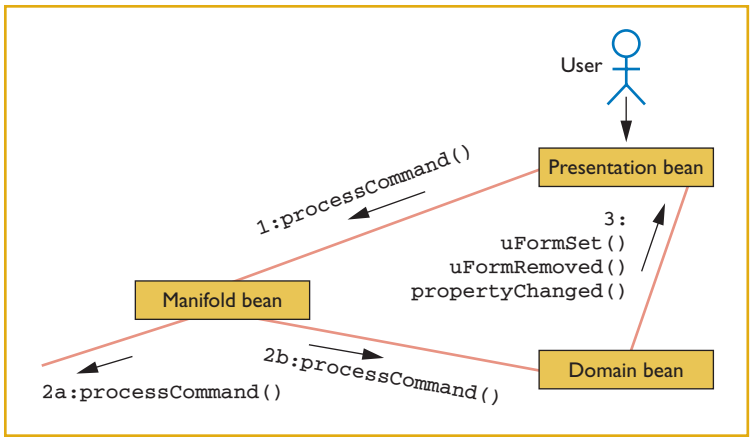

Figure 3. Event exchange and interception in a Manifold bean. Commands generated by the local user affect the domain bean. Action 2a passes the command to the collaboration bus, which broadcasts it to the remote peers.

tools for modifying them. The UForm's graphical representation is called a Glyph, and if a UForm references a subtree of UForms, it corresponds to a composite Glyph or PolyGlyph. Users modify Glyphs via property editors that expose UForm properties, similar to JavaBean property editors, or by directly "grasping" the Glyph and rotating, translating, or otherwise manipulating it. The manipulation process is sampled, and at each step the corresponding primitive UForm operation is generated and passed to the domain tier for execution.

Both property editing and direct manipulation encapsulate one or more primitive UForm operations in a Command object. The Command class implements the Command pattern[8] and must track the argument values to invoke operations on the repositories so that operations can be undone or redone. All Disciple and Manifold components communicate by passing Command messages. The presentation tier of a Manifold bean communicates with the domain tier by passing a Command, as does a Disciple client communicating with another Disciple client or server. The Command messages focus on data manipulation rather than event propagation and thus maintain independence between the clients and the servers.

The domain and presentation beans must be glued together because the execution environments assume single beans. The third bean, the Manifold bean, interacts with the collaboration bus to send and receive collaboration commands. Figure 3 shows how the beans interact. The Manifold bean dispatches a command either to the local domain bean and the collaboration bus simultaneously or first to the bus and then locally after the command is reflected back, depending on the concurrency control algorithm used (optimistic or pessimistic).

Implementing presentation and domain as distinct beans rather than the whole package as a single bean permits mixing and matching more or less complex beans for each. Domain beans can implement complex behaviors and the presentation beans can implement visualizations with varying degrees of realism. Depending on the context, users can download different domain/presentation pairs on demand. The control module (a Manifold bean equipped with the information provided by Disciple) downloads the domain bean based on the computing and network environment characteristics.

XML Data Transformations

Visualizing objects on devices with different computing and communication capabilities requires some content transformation. Desktop clients can represent objects in great detail, with each part represented as an independent object, whereas handheld wireless clients will likely permit only simple representations. If some clients visualize objects in 3D and others in 2D, the 3D coordinates must be converted into 2D, and vice versa. The simple solution of just discarding or adding a z coordinate would not work because the rotations around the x or y-axis would be difficult to handle. Multimedia objects such as images and video represent a particular problem.

Figure 4 illustrates the basic process of data adaptation. Let's assume the user interaction at client α results in a command c_α carrying the information about the user's activities. A command consists of a sequence of primitive operations on the UForm graph G:

- (Op_1), create a vertex u_i,
- (Op_2), delete a vertex u_j, and
- (Op_3), modify vertex property.

The commands must be transformed to a platform-specific format before being delivered remotely.

The transformation $f_{\alpha\beta}$ renders the command c_α to c_β for delivery to site β in two stages. First, the transformer decides whether a particular primitive operation in the command applies at the remote site. The operation will not apply if the remote site's local repository does not contain the particular UForm. The decision is binary: true or false. For example, if c_α is as follows:

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

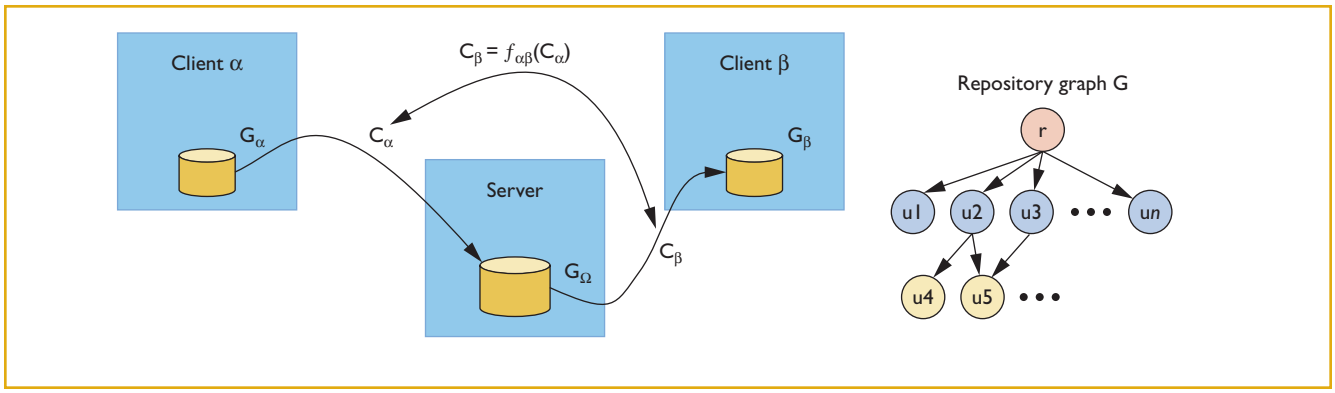

Figure 4. Command transformations to adapt data to client profiles. The structure of the repository graph G with UForms u_i is on the right.

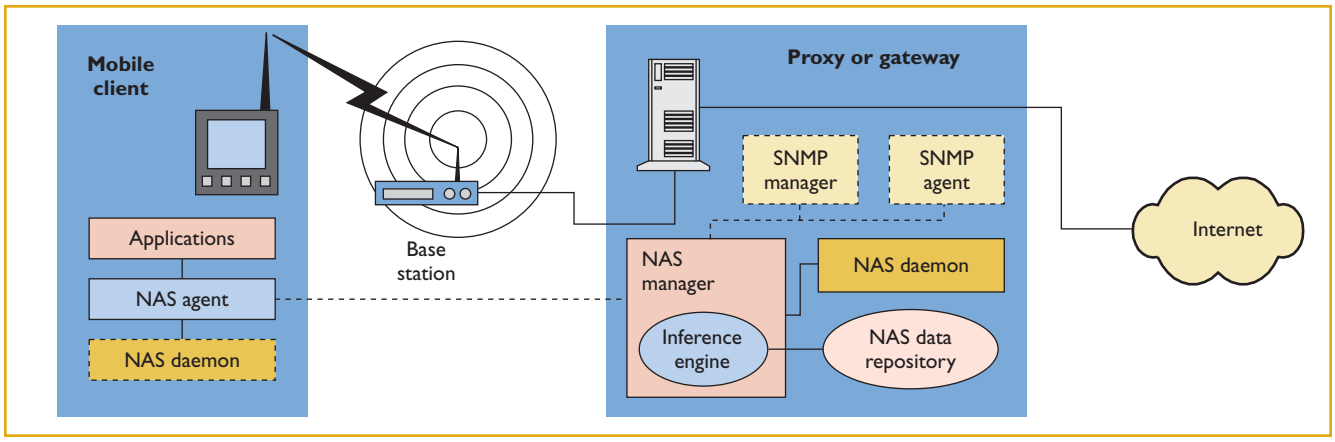

Figure 5. Network Awareness Service architecture. The awareness tasks are distributed from the NAS agent at the mobile host to the NAS manager at the wireless proxy/gateway, thus preserving wireless bandwidth and computing resources.

then cβ might be

$$c^1\beta = f^1\alpha\beta(c\alpha) = Op_1 \circ Op_3 \circ Op_1 \circ Op_3$$

The decisions follow rules, programmed into the data adaptation agents, which reflect different users' interests and platform capabilities.

In the second stage, the properties associated with type Op_3 operations must be converted to match the remote domain. For example, Java for small devices (J2ME) does not support floating-point numbers, so the coordinates must be converted to integer numbers, making it impossible to keep the same data format on all clients. The final result will be

$$c\beta = f^2\alpha\beta(c^1\beta) = Op^{\beta}_1 \circ Op^{\beta}_3 \circ Op^{\beta}_1 \circ Op^{\beta}_3$$

The data adaptation agents transform the data according to user- or developer-specified XSL rules, and the transformer uses the XSLT processor for information transformation. An XSLT-based transformer has the significant advantage of being compliant with the XML markup language. We also plan to consider expert-system-based agents for more powerful rules description and possible automatic learning.

Maintaining consistency between the distributed repositories represents a graph isomorphism problem. Due to the platform-specific adaptation, the graphs are not isomorphic. In a general case, a client state is consistent if its repository graph is approximately isomorphic with a subgraph of the server's repository graph. Our current research focuses on determining the exact conditions for mapping $f\alpha\beta$ and $f\alpha\beta^{-1}$ to maintain state (repository) consistency.

Intelligent Agents and Network Awareness

To deploy this data-centric scheme, the system must build the environment profiles, a task for intelligent

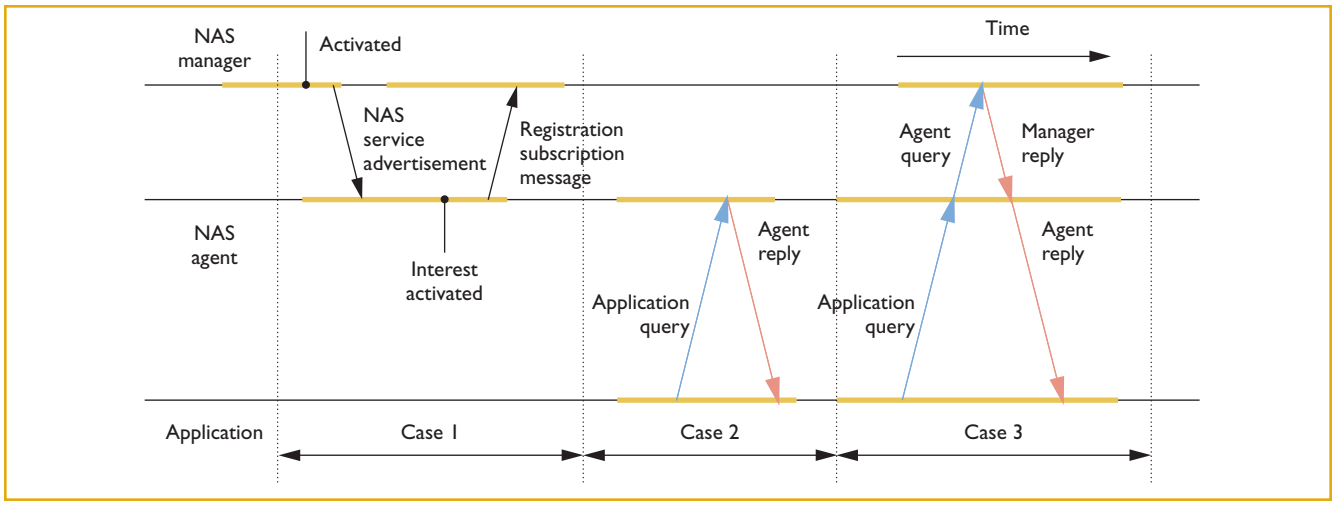

Figure 6. Communication procedure between network applications and network awareness agent.

agents. As Figure 1 showed, agents span all architectural layers because they need information from each to make intelligent decisions. Agents sense the networking environment and determine how to transform information exchanged between conferees to match diverse data representation domains and computing and network capabilities. Agents also dynamically determine Manifold bean composition based on client device capabilities and other profile components.

Intelligent agents determine communication link characteristics, for example, by measuring throughput or wireless link transmission quality. The Disciple framework then automatically adjusts to the environment by applying different information transformations for data adaptation. A brief overview of our network awareness architecture will show how network applications can be optimized for heterogeneous communication environments where both wired and wireless links exist. Further details are available elsewhere.[7]

Network Awareness Architecture

Adaptive network applications need to be aware of network parameters. Figure 5 shows a network-awareness service (NAS) designed to provide such information to applications. It uses four main components: the NAS daemon, NAS data repository, NAS manager, and NAS agent. Unlike a traditional end-to-end network awareness framework in which measurement tasks occur solely at mobile hosts, the NAS framework measures end-to-end network conditions piece by piece.

Network awareness includes discernment of the network's mobile wireless and fixed wired segments. The NAS manager or NAS agent integrates the intermediate measurements on these two parts and provides the final result to the application. The NAS agent can implement different awareness techniques or delegate the measurements to other agents, such as the SNMP (simple network management protocol) agent. The minimum configuration contains only the inference engine, making the architecture lightweight and deployable to small devices.

This piecewise NAS framework is useful because it not only distributes the NAS components between the mobile device and the proxy/gateway but also shifts the awareness tasks from the NAS agent at the mobile host to the NAS manager at the proxy/gateway. This preserves wireless bandwidth and computing resources, a feat proven by analytical and experimental results.[7]

Disciple deploys network awareness by querying the network awareness agent according to the application bean's QoS requirements or by subscribing to the information broadcast by the agent. Figure 6 illustrates the generic procedure for agent-application interaction, which can be used in network application startup and runtime phases. In the startup phase it is used in the proactive mode, while in the runtime phase it can be used in both proactive and reactive modes.

Bandwidth Awareness Example

As an example, let's look at how Disciple characterizes the communication channel by measuring the available bandwidth. We conceptualize available bandwidth as the bottleneck link bandwidth; Disciple therefore uses interpacket spacing to estimate the bottleneck link's characteristics. If two packets travel together and are queued as a contiguous pair at the bottleneck link, their interpacket spacing is proportional to the processing

time required for the bottleneck link to transmit the second packet. We modified a technique presented elsewhere[9] to apply it to asymmetric wireless links as well.[7]

Information on available bandwidth proves important in an application presented below, Palmscape. The wireless cellular link that connects Palm Pilots to the network has a narrow bandwidth and large latencies (2–15 seconds), which makes it impossible to broadcast direct object manipulation events to such collaborators. Disciple queries the WAS framework about the link bandwidth and delay and, if they appear unfavorable, buffers the manipulation events for such links. The collaborators see only a busy icon for the manipulated object. When the manipulation is completed, Disciple delivers the buffered events to the collaborators so they can see an animation of what happened during the manipulation.

Testing Disciple and Manifold

In testing our framework, we sought to answer two practical questions:

- How well does it generalize to arbitrary applications?
- Can users collaborate equitably?

Manifold offers a lightweight, scalable, extensible architecture for building any collaborative application that operates as a generalized structured data editor. The application logic needed to edit the data repositories remains the same across tasks and platforms. Whereas data parameters such as amount, type, fidelity, and accuracy differ across domain beans for disparate platforms, the processing machinery remains consistent, and most Java classes are reused across platforms.

Applications implemented and tested to date include text-based chat, whiteboard, 3D collaborative virtual environment (CVE), collaborative situation map, speech signal acquisition and processing, image analysis, and a medical image-guided diagnosis system for the diagnosis of leukemia. The applications sampled below use Manifold to create graphics editors with 2D versus 3D object representations.

Trial Applications

The cWorld JavaBean, developed using Java3D, enables synchronous, multiuser creation of collaborative virtual environments (CVEs), as shown in Figure 7. CWorld does not require special hardware and can be operated via keyboard and mouse; it also supports the Magellan Space Mouse that provides the six-degrees-of-freedom movement used for navigating 3D spaces.

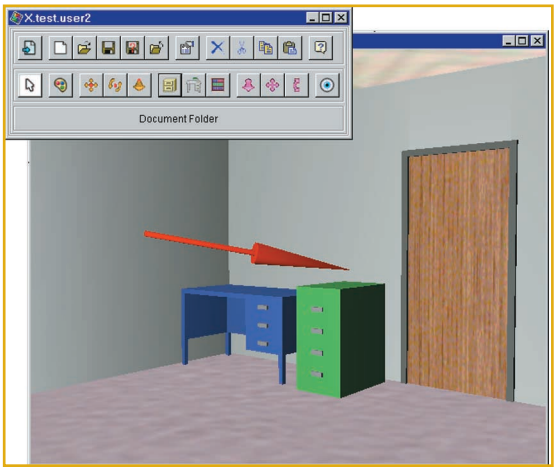

Figure 7. Sample collaborative virtual environment (CVE), built using cWorld. The 3D telepointer follows the user's current location and line of sight.

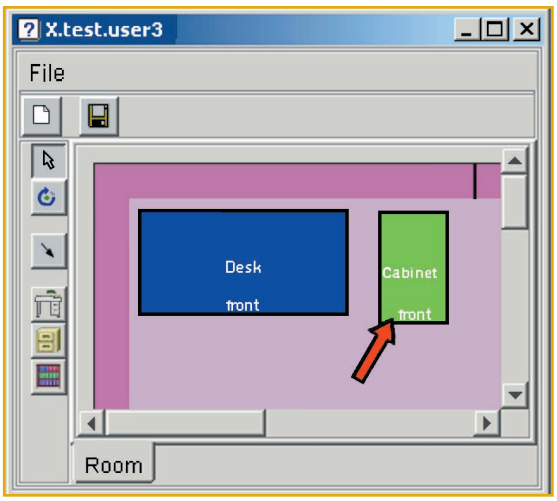

Figure 8. Floor plan rendered in Flatscape. This plan corresponds to the room shown in Figure 7; the 2D telepointer follows the user's view.

CWorld performs the following primitive operations on the UForm tree:

- Op_1 creates simple geometric figures (box, cylinder, cone, sphere) and lights (directional, point, spotlight). The furniture objects (desk, cabinet, bookcase), which are composite UForms, can be imported from files. The telepointer is also a composite UForm (cylinder + cone).

- Op_2 deletes any of the above-listed UForms.
- Op_3 sets UForm properties such as color, texture, position, dimensions, and manipulation constraints (for example, furniture objects cannot "fly"). The lights have additional properties such as direction, attenuation, and spread angle. The user can also apply 3D affine transformations to UForms by manipulating the corresponding Glyphs or PolyGlyphs.

User commands result in one or more primitive operations. For example, moving a figure to background or foreground, or grouping or ungrouping figures, requires several primitive operations.

Each user has a unique 3D telepointer (Figure 7) that functions as a primitive avatar and appears at the user's discretion. The telepointer reflects the position and orientation of the user's line of sight.

Figure 8 shows another application bean, the Flatscape editor, which we developed using Java2D. As an example, the UForm for the desk shown in Figure 8 has the following characteristics:

- ID: 92600832
- type: "desk"
- file: "icons/desk.gif"
- width: "51"
- height: "28"

Flatscape operations on the UForm tree include

- Op_1 creates simple geometric figures (line, rectangle, ellipse, polygonal line, spline) and bitmap images. The figures can be grouped into composite UForms.
- Op_2 deletes any of the above-listed UForms.
- Op_3 sets UForm properties such as contour color, thickness, and dash; fill color; position; dimensions; visibility; and manipulation constraints (for example, movement only along the x-axis). The lines also have arrow styles, and the splines have other properties. The user can apply 2D affine transformations to UForms by manipulating the corresponding Glyphs or PolyGlyphs.

Flatscape telepointers appear as 2D arrows pointing toward the user's current view. This differs from typical telepointers, such as that found in GroupKit (see the sidebar on page 27), that show only position and not orientation. In our 2D implementation, the telepointer's owner, not the recipient, controls its visibility.

Using J2ME CLDC 1.0 (Java 2 Micro Edition — Connected, Limited Device Configuration), we developed a simpler version of Flatscape that runs on PalmPilots. Palmscape, as it is called, has most of Flatscape's basic functionality (create, delete, move, and rotate objects, for example) but accepts user input via the Palm stylus and buttons.

Human Factors Experiments

How does device heterogeneity affect human performance?

We explored this question using an instantiation of this approach in a furniture arrangement application. The application presents collaborators' task views consistent with their platform capabilities. We conducted experiments to measure how display differences would affect collaboration.[10] The results showed that, rather than being hindered by platform differences, a reduced-display user (2D) performed well in collaboration with an information-rich display user (3D). We also observed that users are willing to sacrifice speed and visualization fidelity in exchange for universal access and timeliness of information.

Future Work

By abstracting the collaborative tasks as editing the data repositories, our approach to synchronous collaboration preserves the application logic across platforms and tasks. The main difference across groupware applications is in the presentation logic. We address this difference, along with differences in the visualization and interaction tools available on diverse platforms, by custom-designing a presentation module for each platform.

The current Disciple framework lacks such features as group awareness, concurrency control, and access control. The design and implementation of these features will likely follow the pattern established thus far: consistent application logic across platforms combined with platform-specific presentation logic. Application-level proxies could complement our approach for nonstructured data (such as images and video), and we plan to explore this in the future. Also, because mobile device CPU and memory resources will likely outpace their display and network capabilities, stand-alone full applications are feasible on small devices. Because our approach — unlike a proxy-based approach — does not require continuous connectivity, it lets the user continue working even when the link goes down temporarily. We may include proxies for data transformation but the application logic always resides on a client rather than on its proxy.

Some issues we've addressed here pertain also to pervasive or ubiquitous computing applications that must be downloaded and run on different platforms. Again, our focus on data and communication protocols liberates developers to tailor applications to particular access device.

Our continuing work concentrates on developing more real-world applications while improving performance of data transformation and repository synchronization, network monitoring, and data adaptation. For more information about the Disciple project and heterogeneous collaboration, see the Web site at http://www.caip.rutgers.edu/disciple/.

Acknowledgments

Research contributors to this project include professors James Flanagan, Marilyn Tremaine, and Allan Meng Krebs, and graduate students Mihail Ionescu, Liang Cheng, and Bogdan Dorohonceanu. The research is supported by NSF KDI contract no. IIS-98-72995 and DARPA contract no. N66001-96-C-8510 and by the Rutgers Center for Advanced Information Processing (CAIP) and its corporate affiliates.

References

1. I. Marsic, "DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments," *ACM Comp. Surveys*, vol. 31, no. 2es, June 1999.
2. I. Marsic, "An Architecture for Heterogeneous Groupware Applications," *Proc. 23rd IEEE/ACM Int'l Conf. Software Engineering* (ICSE 2001), IEEE Press, Piscataway, N.J., May 2001, pp. 475-484.
3. World Wide Web Consortium, Extensible Markup Language (XML) home page, http://www.w3.org/XML/.
4. W3C, Extensible Stylesheet Language (XSL) home page, http://www.w3.org/Style/XSL/.
5. S.A. Roth et al., "Visage: A User Interface Environment for Exploring Information," *Proc. Information Visualization*, IEEE Press, Piscataway, N.J., 1996, pp. 3-12; available online at http://www.maya.com/visage/base/technical.html.
6. C. Francu and I. Marsic, "An Advanced Communication Toolkit for Implementing the Broker Pattern," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems* (ICDCS 99), IEEE Press, Piscataway, N.J., 1999, pp. 458-467.
7. L. Cheng and I. Marsic, "Piecewise Framework for End-to-End Network Awareness Service in Heterogeneous Data Networks," submitted for publication.
8. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
9. R.L. Carter and M.E. Crovella, "Measuring Bottleneck Link Speed in Packet-Switched Networks," tech. report BU-CS-96-006, Computer Science Dept., Boston Univ., Mar. 1996.
10. I. Marsic et al., "Designing and Examining PC-to-Palm Collaboration," to appear in *Proc. 35th Hawaiian Int'l Conf. System Sciences* (HICSS-35), IEEE Press, Piscataway, N.J., Jan. 2002.

Ivan Marsic is an assistant professor of electrical and computer engineering at Rutgers University. He is chief architect of the Disciple system. His current research interests include groupware, mobile computing, computer networks, and human-computer interfaces. Marsic is a member of the IEEE and the ACM and has been a consultant to industry and government.

Readers can contact the author at marsic@caip.rutgers.edu.