Middleware for Reliable Mobile Medical Workflow Support in Disaster Settings

Steven W. Brown, MS[1] William G. Griswold, PhD[1,2]
Barry Demchak, BA[2] Leslie A. Lenert, MD[1,3]
[1]California Institute for Telecommunications and Information Technology and
[2]Computer Science and Engineering
University of California, San Diego, La Jolla, CA
[3]Veterans Affairs San Diego Healthcare System, San Diego, CA

Mobile information technology can help first responders assist patients more quickly, reliably, and safely, while focusing resources on those most in need. Yet the disaster setting complicates reliable networked computing. The WIISARD system provides mobile IT support for medical response in disasters. Cached remote objects (CROs) are shared via publish/subscribe, enabling disconnected operation when out of network range and ensuring data consistency across clients with rollback/replay. CROs also provide a flexible, familiar, and performant programming model for client programmers. A drill with the San Diego MMST showed that a basic CRO-based client-server architecture is insufficient, because prolonged network failures—to be expected in disaster reponse—inhibit group work. We describe an extension of the CRO model to clusters of computers that supports group work during network failures.

INTRODUCTION

The standard top-down command-and-control hierarchy used for medical response in disasters (MRiD) provides for reliable and safe on-site diagnosis, treatment, and transport of patients to hospitals. As motivated below, it is also slow and information poor, putting some patients at risk.

Mobile information technology holds the promise to ease communication and information management, enabling first responders to reliably and safely treat more patients and focus resources on those most in need. Yet, the use of IT in a disaster setting is fraught with challenges, and a failure that requires responders to revert to their old methods is unacceptable.

This paper describes the WIISARD client-server architecture and middleware for mobile IT support for MRiD. It employs remote objects to resolve many of the problems that arise in IT for MRiD. Remote objects (1, Ch. 5), with local caching provide a flexible, convenient, and familiar programming model for application programmers, hiding the complexities

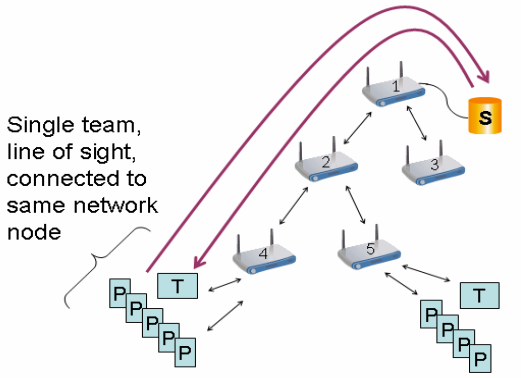

Figure 1. The WIISARD system architecture. T = Tablet PC, P = PDA, and S = server.

of remote data access, disconnected operation, and data consistency management.

The WIISARD system (see wiisard.org) was tested in a disaster drill performed by the San Diego Metropolitan Medical Response System teams at the Del Mar Fairgrounds on Nov. 15, 2005. 100 patients were triaged and tracked with WIISARD. The ability to monitor patient status at a distance was welcomed by responders and the medical command. Yet, support for work groups was compromised during network partitions, pointing to an improved design.

This paper contributes a feasibility study for supporting coordinated teams with a client-server architecture in the harsh conditions of a disaster. We further identify system requirements and methods for successful mobile IT support for coordinated teams.

BACKGROUND AND REQUIREMENTS

The initial response to a disaster at a site secures the scene. Medical teams then move in, establishing areas for triaging patients, decontamination (if necessary), subsequent treatment, and transport to area hospitals. The basic workflow is a pipeline, with patients moving through the stages in succession.

The standard information tools of response include: Simple Triage Rapid Treatment (START) tags that record each patient's condition and any critical treat-

ments (such as Mark I kits for nerve agents); clipboards for supervisors to track information such as transport logistics; and whiteboards and easels for commanders to track the status of the response. Information is moved among these tools by word of mouth, often by radio. This system of information management is mature, but slow and information-poor. Patients in need of immediate help may be just out of sight, dangers are not apparent, and key trends may be invisible to the command.

Mobile IT in the disaster setting can automatically propagate information among responders and construct information displays appropriate to each responder's role (e.g., triage provider, triage supervisor, medical command, etc.). With the WIISARD system (2), providers carry wireless personal digital assistants (PDAs) with integrated barcode scanners, supervisors carry Tablet PCs, and medical command employs large-display devices. Patients are tagged with a wireless smart tag (3) or a traditional paper START tag. Network communication is supported by a portable 802.11b mesh network (4).

A triage provider, for example, places a tag on the patient, scans the tag's barcode, and then, assessing the patient, clicks several items on a screen that looks like a START tag. (If just setting the patient's status, then triage can be performed using buttons on the smart tag.) The patient's information is automatically distributed, for example to the triage supervisor's patient list and the medical command's graphs and map display. No pause for human conversation is necessary, and the communication is essentially instantaneous. START triage can be completed in about 30 seconds, roughly half the time of the traditional way. The smart patient tag continuously updates the patient's location, and if connected to a pulse/oximiter, continuously monitors patient status.

Because of the harsh dynamics of the disaster scene, devices will fail (even if deployed in housings or as ruggedized variants), networking will be interrupted, and providers or patients will walk out of the network's range. For responders to depend on this infrastructure, it cannot fail, but rather must degrade services gracefully in response to underlying failures.

RELATED WORK

Others have explored the use of handheld computers in triage situations. To cope with difficult communication issues in battlefield, the Army's Battlefield Medical Information System Telemedicine-Joint (BMIST-J) forgoes network connectivity in the field, relying on a memory cache on the patient (in a "smart" dog tag) to carry limited field data (5). IRIVE is a triage and medical data collection system that fields asimple client-server architecture (6). The Automated Remote Triage and Emergency Management Information System (ARTEMIS) transmits START data to a central repository, using publish/subscribe (discussed later) and local caching to distribute and manage data (7). It is like the WIISARD system in structure and could take advantage of the contributions of this paper.

THE WIISARD DESIGN

WIISARD employs a client-server architecture and supporting middleware substrate. We envisioned a scenario where a single provider would use a handheld device to collect and transmit START data to a central server for access by others. Because the server resides in a support vehicle, is connected by a wire to a mesh network node, and can be replicated, data loss is rendered unlikely (See Figure 1). (The support vehicle also stores spare field devices, provides power generation for the recharging of spare batteries, etc.) So that a client can tolerate its own loss of network, we decided that each client would hold a local copy of the data it uses (in essence, a write-through cache). Should a client become disconnected, the provider might be alerted, but could continue working uninterrupted on the client's copy of the data. When the network connection is restored, modified local data is pushed up to the server, and modified remote data is pushed down to the client.

This simple design creates two challenges. First, while a client is disconnected and modifying its data, other clients may be modifying the same data, unbeknownst to each other. When the network connection is re-established, a single "true" copy of the modified data must be produced and pushed to each client. Simply locking all devices that are sharing data until a change can be committed would be slow and could freeze the system indefinitely. Second, the details of managing network connections, object transport, caching, and automated reconciliation of the conflicting updates are daunting and subtle, so they should be solved once in the middleware and be largely transparent to client application programmers.

Our solution employs four elements: remote objects, publish/subscribe (8), rollback/replay conflict resolution, and an object abstraction middleware layer that hides details from clients (See Figure 2).

Cached Remote Objects (CROs)

A central data definition (schema) at the server constitutes the definition of object types (classes) that comprise the data (objects) of the system. The home for objects is the server; the objects present on a client device are merely copies, perhaps out of date or locally modified. The server represents and enforces the data truth of the entire system. Each object type is given modification semantics. For each method defined on the type that modifies its object's state, a *command* determines what actions are to occur to reflect the state change back to the server. Each

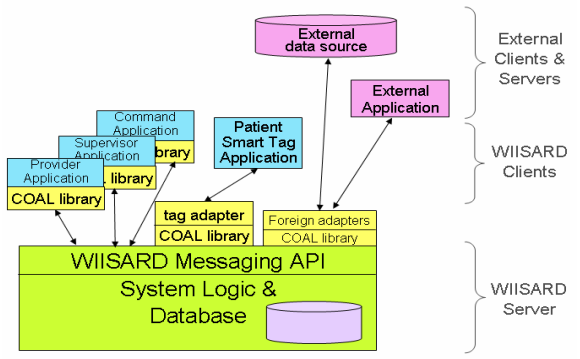

Figure 2. The WIISARD client-server architecture.

command has precondition checks for committing its state change back to the server.

A trivial CRO class can be automatically generated such that each object field is accessible through generated getter/setter methods. The default command for the setters is discussed in the next subsection; the default precondition check is discussed in the following subsection on rollback/replay.

Publish/Subscribe Communication Protocol

The server makes the system's data available by *publishing* objects on the network. For a client device to get a copy of an object from the server, say a Patient object, it *subscribes* to the object using a reference (pointer or id) to the object, *Subscribe(ref(X))*. To bootstrap this process—prior to the client having any object references—the client can subscribe to an object type itself: *Subscribe("Patient")*. This gives the client access to an array of references to all objects of the given type. The client can then subscribe to the individual objects through these references.

Publish/subscribe (*pub/sub*) is event-based. A client and server communicate via non-blocking asynchronous messages rather than synchronous, blocking procedure calls. Consider a client modifying the Status field of a Patient object through its setStatus setter method. The default command for the setter transparently sends an asynchronous message (event) *PatientSetStatus(ref(X), newStatus)* to the server. Meanwhile, the client continues computing on the modified object. The server formally accepts this modification by sending a matching *success* message back to the originating client, and all subscribers to that object receive a *PatientModified(X')* event containing a copy of the modified object, causing the local copy of the object to be updated and an internal *modified(ref(X))* event to be sent to the client application. Normally, the client will have *registered* an action for such an event, say for redrawing active displays of the object. This registration is just a local subscription for performing a local action rather than data movement across the network.

The non-blocking nature of this protocol enables responders to continue working even as others access the same data. Also, when a client is disconnected from the network, server-bound events are queued, and the client continues operation on the locally modified objects. Upon reconnection, the queued events are sent out, and queued-up modification events from the server arrive, updating local copies of objects. The architecture is thus naturally tolerant of contention, high latency, and network loss.

Conflict Resolution via Rollback/Replay

There is always the chance for modification conflicts, especially after a lost connection is restored: two clients each contemporaneously modify the Status field of their local copy of the same Patient object, and the setter sends a *PatientSetStatus* event back to the server. It is likely that one of these modification events will be applied to the object and the other will be rejected. For example, the default precondition for committing a field modification is that it can be applied only if the server and client agree on the field's old value—in other words, that the client is modifying what it thinks it's modifying.

To handle failures, each client's event queue holds not only outgoing events, but also events as yet unacknowledged by the server (called *pending*). Each queued event holds a reference to the modified object (called the *predicted object*), the new values for the modified fields (for updating on the server), as well as a copy of the previous values of the modified fields (for local roll back). When a *failure* message is returned for a pending *PatientSetStatus* event, its modifications and that of all subsequent pending modified events are "rolled back"—undone—using the reference and the saved previous values. Then the effect of the arriving *PatientModified(X')* event generated from the "winning" client's *PatientSetStatus* event is applied. Finally, the rolled-back events are "replayed"—their modifications are re-applied and the events re-queued. It is possible that some or all of the replay will fail, as a replayed event could depend on the value produced by the failed event. In this case, the replay is stopped at the failed event and a pop-up window is raised to the client's user, asking whether to proceed with the replay anyway, abort the remainder of the replay, or substitute a value for the failed event and then continue replay. This approach closely matches work by Chang and Curtis on mobile access to object databases (9).

Cached Object Abstraction Layer (COAL)

These complexities are best hidden to ease the development of client applications. Most of the unique behaviors of CROs are hidden in their private method bodies and the underlying COAL middleware runtime library, constituting a reusable, transparent abstraction layer. Only subscription to objects is unusual. The client application's registration of actions that respond to object updates is not unusual. In practice,

this is similar to the handling of modifications performed locally, say via the user interface.

The COAL middleware concept is flexible. Since the WIISARD patient tags have limited memory, a "tag adapter" was developed as an intermediary. It links to the COAL and maintains CROs for all the tags, using a simple data protocol to communicate with the tags. The tag adapter runs on the server machine as a standalone program, although it looks like any other COAL client to the server software.

Evaluation

During the drill described in the introduction, 6126 modification events were generated on 100 patients, plus medications, ambulances, and hospitals. A typical event is 64 bytes. Non-network overhead for sending a message was under a millisecond. Although there were 70 devices deployed, there were 610 TCP connections established to the WIISARD server during the drill. This does not count failed attempts, which were presumably numerous, if the network environment resulted in over 500 disconnects. Yet, complaints from the responders were few, except for one long network partition, described below. The general responder perception was instantaneous response, presumably due to WIISARD's asynchronous CROs and disconnected operation.

Event conflicts were rare at 1.3%, despite the conflict opportunities created by the network problems. Most conflicts were due to data entry errors caused by a flaw in our client UI. To compute the costs of rollback/replay, we ran a test of 100,000 individual rollback/replay event sequences, showing that the rollback and replay of a single event on an iPAQ 5555 PDA running Linux takes 200 microseconds.

A prolonged network partition occurred during the drill. (in Figure 1, suppose that the network was partitioned between nodes 2 and 4, cutting the bottom-left team off from the server.) Even though the system performed as intended, our approach to disconnected operation was inadequate. A triage supervisor reported frustration that although he was standing next to a working triage provider, the provider's data did not immediately appear on his device (because the provider's events were locally queued for transmission to the server). Thus, the triage provider could continue to work, but the supervisor's work flow was interrupted, putting patients at risk. Yet, there was little need for a triage provider to see a transport provider's data. Teams, primarily, need to stay in continuous contact.

EXTENSION OF ARCHITECTURE

To address the communication needs of teams, we extended the CRO concept to include a team-level cache of objects. We implement this by having each wireless node in the network host a local mirror of the main server. Normally, a local server passes all requests through and caches an image of the database. A partition causes the local server attached to the highest node in the mesh's routing tree to become the local server for all clients in the partition. Thus, data continues to be shared between proximate responders, presumably comprising one or more teams.

The question then is how to commit changes made on a local server back to the main server upon reconnection. The local server cannot do it, because resolving some conflicts requires responder judgment. Thus, we chose to have clients to keep all their events in the queue while disconnected from the main server, permitting a full rollback/replay on the client. We extended work on two-level commits—locally synchronous and globally asynchronous (10)—to support *two levels of asynchronous commits*. Asynchronous access to the local server avoids blocking operations over the network. This multi-level mechanism has not yet been developed. Given this mechanism's similarity to the basic one, though, we expect performance to be similar. It operates as follows.

Modifications to the local server's objects are only *locally committed*. Unlike globally committed events, locally committed events stay in a client's event queue until reconnection with the main server, at which point they behave like pending events in that they are sent to the main server for reconciliation. The use of local commits resolves local conflicts in a timely manner, and most would not require conflict resolution on the main server, since data sharing across teams is less frequent than within teams, and tends to be write-read sharing, not write-write.

If a global commit fails for a client's locally-committed change, the resolution proceeds as described in the previous section for a failed pending change. The only difference is that a locally-committed event (and its predicted object) may be held by multiple clients, not just the client that created the event. Each client will perform its local replay with the "winning" event, but may fail at unique places in their local replay, depending on their local event queues and how each depends on the state created by the winning event. It is possible that—unlike with the resolution of pending conflicts—the resolution of locally-committed conflicts would be visible to other clients, thus causing conflict/replay ripples to these other clients. However, the event generated by manually resolving a conflict is no different than any other. In particular, these affected clients are not blocking on this resolution event. Each is equally able to asynchronously take up the conflict and attempt resolve it first.

The computational demands on the client are no more with this approach than with our basic scheme, but the prospect of prolonged disconnects creates the possibility for costly reconnects. Time can be saved

by submitting commits in batches rather than serializing them. It is also possible to declare a command that modifies rapidly changing fields (e.g., location) as *volatile.* Volatile commands of the same type can be combined in the event queue by updating the old event with the new one's data, rather than queued. A command can be declared volatile only if the consumers of the fields in question can tolerate a lack of ordering. A map display may be fine if updated only with the last received location of a patient. Seeing every patient status change may be critical.

CONCLUSION

The demands on a mobile system for disaster response are unremitting, but a few concepts help: cached remote objects distributed over pub/sub and kept consistent through rollback/replay. A middleware layer hides these complexities with familiar object-oriented programming abstractions.

For mobile IT to support team efforts under a network partition, team-level object caching is required. A server hierarchy can be laid over the network to tolerate partitions—a team can still work together, linked through their common mesh node's local server. Cost-effective rollback/replay is preserved by providing two levels of asynchronous commits and reducing replay costs through batching and customized handling of volatile objects. From our experience, we propose this principle:

> ***System structure principle for medical response in disasters***: *A disaster response system's structure should follow human organizational structure. In particular, system failures should behave like organizational failures, causing uncoordinated response rather than work stoppage.*

Without such architectural accommodations, a disaster response system will only be able to support *individual* provider efforts, and perhaps *command center* efforts where completely accurate data is not always necessary for decision making.

ACKNOWLEDGEMENTS

Supported by National Library of Medicine contract N01-LM-3-3511. We thank the WIISARD team, expecially R. Huang and F. Liu for their client work.

REFERENCES

1. Coulouris G, Dollimore J, Kindberg J. *Distributed Systems Concepts and Design*, 4th ed. Addison-Wesley; 2005, p. 944.

2. Chan TC, Killeen J, Griswold W, Lenert L. Information technology and emergency medical care during disasters. *Acad Emerg Med.* 2004; 11(11):1229-36.

3. Palmer DA, Rao R, Lenert LA. "An 802.11 Wireless Blood Pulse-Oximetry System for Medical Response to Disasters", *Amer. Med. Inf. Assoc. 2005 Annual Symp.*; 2005.

4. Arisoylu M, Mishra R, Rao R, Lenert L. "802.11 Wirless Infrastructure To Enhance Medical Response to Disasters", *Amer. Med. Inf. Assoc. 2005 Annual Symp.*; 2005.

5. Battlefield Medical Information System Telemedicine-Joint, website, https://www.mc4.army.mil/BMIST-J.asp.

6. Gaynor M, Seltzer M, Moulton S, Freedman J. "A Dynamic, Data-Driven, Decision Support System for Emergency Medical Services", *Int'l Conf. on Computational Sci.*, Springer-Verlag; 2005, p. 703-11.

7. McGrath SP, Grigg E, Wendelken S, Blike G, De Rosa M, Fiske A, Gray R. "ARTEMIS: A Vision for Remote Triage and Emergency Management Information Integration", unpublished manuscript, http://www.ists.dartmouth.edu/projects/frsensors/artemis/papers/ARTEMIS.pdf.

8. Eugster PT, Felber PA, Guerraoui R, Kermarrec R. "The many faces of publish/subscribe", *ACM Comput. Surv.*; 2003 Jun;35(2):114-31.

9. Chang S, Curtis D. "An Approach to Disconnected Operation in an Object-Oriented Database", *3rd Int'l Conf. on Mobile Data Mgmt*; 2002 Jan.

10. Pitoura E, Bhargava B. "Data Consistency in Intermittently Connected Distributed Systems", *IEEE Trans. on Knowledge and Data Eng.*; 1999 Nov;11(6):896-915.