GT/SD: Performance and Simplicity in a Groupware Toolkit

Brian de Alwis and Carl Gutwin
Department of Computer Science
University of Saskatchewan
110 Science Place, Saskatoon, Canada
bsd@acm.org, carl.gutwin@usask.ca

Saul Greenberg
Department of Computer Science
University of Calgary
2500 University Drive NW, Calgary, Canada
saul.greenberg@ucalgary.ca

ABSTRACT

Many tools exist for developing real-time distributed groupware, but most of these tools focus primarily on the performance of the resulting system, or on simplifying the development process. There is a need, however, for groupware that is both easy to build and that performs well on real-world networks. To better support this combination, we present a new toolkit called GT/SD. It combines new and existing solutions to address the problems of real-world network performance without sacrificing the simple programming approach needed for rapid prototyping. GT/SD builds on the successes of earlier groupware toolkits and game networking libraries, and implements seven ideas that help solve problems of network delay, quality of service, rapid development, flexibility, and testing.

Categories and Subject Descriptors

H.5.2. [Information Interfaces]: User Interfaces—*prototyping*. D.2.2 [Software Engineering]: Design Tools and Techniques—*user interfaces*.

General Terms

Performance, Design, Human Factors

Keywords

Toolkits, network programming, groupware, extensibility.

1. INTRODUCTION

Real-time distributed groupware is software that lets people communicate, work, and play together at the same time but from different places. Examples include shared editors, screen-sharing tools, communication applications, and networked multiplayer games. There are now many tools available to help application programmers develop real-time distributed groupware: specific toolkits are available, such as GroupKit [22], JSDT [16], Suite [4], or Fiaa [30]; lower level distributed-object protocols are being built into development environments (e.g., Java RMI or C# Remoting), and several game libraries (e.g., TNL or Raknet) also provide network support.

We can broadly categorize existing tools such as these in terms of how they approach the simplicity-power tradeoff. That is, they either make things easy for the groupware programmer (giving a 'low threshold' to entry), or they provide maximum control and power over detailed aspects of their system (giving 'high ceilings' of expressiveness) [8][25]. Very few tools do both. For example, game toolkits provide a great deal of control over networking issues, but are complex and have a steep learning curve; as a result, they are generally adopted only by skilled developers committed to game production. At the other end, groupware toolkits like GroupKit [22] are easy to learn by average programmers. However, these tools hide most of their internal details. The lack of precise control means that applications may not perform well in real-world settings, and that programmers may hit ceilings that stop their exploration [8][25]. With these tools, programmers are typically restricted to proof-of-concept prototypes.

No existing groupware toolkit takes on the dual challenge of providing control over vital aspects of groupware networking, while still maintaining a simple approach for the application programmer. This middle ground is important for a large number of potential developers and a large number of groupware applications. Added control over networking is vital for any application that is to be used on the Internet or a mobile network. Simplicity is also important because rapid prototyping of new ideas may not warrant full development efforts – for example, for groupware researchers and students who create many different prototypes rather than a long-term product line.

To address this need, we have built a new groupware toolkit called the *Groupware Toolkit / Shared Dictionary* (GT/SD for short). GT/SD focuses on issues of networking performance and simple data sharing. It grows out of our experiences building groupware toolkits [22] and working with game libraries [6], and is designed to support rapid development of groupware that can run successfully on real-world networks. In this paper, we focus on seven ideas that set GT/SD apart in the world of groupware toolkits:

- *Latency-management techniques* to avoid situations where groupware systems exceed available bandwidth;
- *Application-level network control* to provide service levels that match groupware messaging requirements;
- *A generic shared dictionary* to allow rapid prototyping of data-intensive applications;
- *Different types of message content* from basic data types, to custom objects, to very large objects;
- *Multiple messaging paradigms* that let developers use polling, events, shared data, and publish/subscribe;
- *Debugging and testing support* through replay mechanisms and statistics gathering;
- *Extensibility* to enable experimentation and support novel research ideas.

The underlying philosophy of the GT/SD toolkit is that it should be easy to build groupware that performs well on real-world networks. In the next sections we review previous research into groupware toolkits, provide an overview of GT/SD, and then discuss each part of the toolkit.

2. Related Work

GT/SD evolved through our experiences with three areas of previous work: distributed-systems infrastructure, other groupware toolkits, and game networking libraries. Much of this prior work can be organized at a high level by the idea of *distribution transparency* [20], that is, the degree to which the environment hides the details of the distributed system from the application programmer.

2.1 Distributed-systems infrastructure

Networking tools are now a familiar part of many programming languages and environment, as briefly summarized below.

Sockets. BSD Sockets are the *de facto* network programming interface; they provide a low-level, byte-oriented perspective on network communication. Although sockets provide maximum control, they are tedious to use for all but the simplest networking applications. As a result, application programmers often look for higher-level programming abstractions for network communication.

Remote Procedure Call (RPC) mechanisms. Java RMI and C# Remoting are examples of synchronous RPC systems. RPC systems disguise client/server communication as method calls. Although this approach is powerful, the transparent approach of RPC breaks down when network problems occur [16]. The RPC model is also problematic in data sharing situations that do not match the model's semantics or assumptions. RPC systems typically expose only limited parts of their implementation to the programmer.

Middleware. Loosely-coupled communication middleware solutions, such as store-and-forward messaging queues (e.g., IBM MQ series, Java Message Service) offer guaranteed delivery, but sacrifice real-time or near-real-time performance. The reliability and ordering guarantees provided by these systems can be overkill for some groupware requirements (e.g., for telepointer updates).

Distributed-object systems. Industrial-strength approaches, such as CORBA and J2EE, are powerful but usually too heavyweight for the needs of prototype-oriented groupware developers. In particular, programmers require substantial training to use these techniques effectively.

2.2 Groupware toolkits

Previous groupware toolkits have focused on a variety of design approaches and goals. Early toolkits were primarily concerned with simplifying the problems of basic connectivity (e.g., [2]). The second generation of toolkits went beyond network connectivity to provide additional features or explore different design approaches. There were several main themes. For example:

- supporting architectural flexibility to allow experimentation with different architectural styles (e.g., [2]);
- supporting algorithmic flexibility and architectural nuances to experiment with optimization and consistency strategies (e.g., Clock [11] or Prospero [5]);
- highly specific types of groupware, such as the transformation of existing single-user applications for use by groups (e.g., JAMM [1]), or single-display groupware for co-located groups (e.g., SDG toolkit [28]);
- simple development of groupware, including simple programming paradigms (e.g., GroupKit [22]) and provision of groupware widgets (e.g., MAUI [15]);
- investigations of particular groupware features such as interface coupling (e.g., Suite [4]);
- investigations of metaphors for organizing groupware applications, such as a rooms environment [10].

These toolkits emphasize particular themes in developing groupware applications, but none of them focuses on the issues of building groupware that performs well on real-world networks. Performance has been looked at in considerable detail, however, by game libraries, as described below.

2.3 Game networking libraries

Game networking libraries have much in common with real-time groupware, and networked multiplayer games already have a proven record in efficient networking. Multiplayer games are similar to groupware in that they send short, frequent messages that are generated from human interactions with the game, and they send several different types of messages with different requirements for reliability and latency. Thus, a reasonable starting point for improving groupware networking is to learn from games.

Commercial or open game networking libraries such as TNL (opentnl.sourceforge.net), Raknet (rakkarsoft.com), or Zoidcom (www.zoidcom.com) provide a number of techniques for improving the performance of real-time network-based games. As analysed by Dyck et. al., [6], these libraries provide three main types of support:

- *bandwidth reduction techniques* includes methods for encoding and compressing data, for rate and flow control, and aggregating messages;
- *reliability and ordering techniques* includes multiple reliability levels and message-level reliability;
- *latency reduction techniques* include streams with different order requirements, lazy state data policies, and quickest-delivery policies for critical data.

Although game libraries provide a great deal of control over network performance, these libraries are not well suited to the needs of groupware developers for several reasons. First, game libraries are complex and difficult to learn, with large APIs and unintuitive programming conventions. Second, game libraries often require that application programmers understand network concepts (such as loss patterns) or implementation details (such as Nagle's algorithm or socket buffer sizes) in order to use the tools. Third, game libraries are set up for the needs of games, and often provide capabilities (or enforce programming styles) that are not needed or are at odds with typical kinds of groupware such as shared editors, communication systems, or more casual games.

We now turn to GT/SD. While certain aspects of GT/SD are new, its primary contribution is in synthesizing techniques and lessons from previous groupware toolkits and from game libraries into a single system that serves to improve network performance without unduly sacrificing development simplicity.

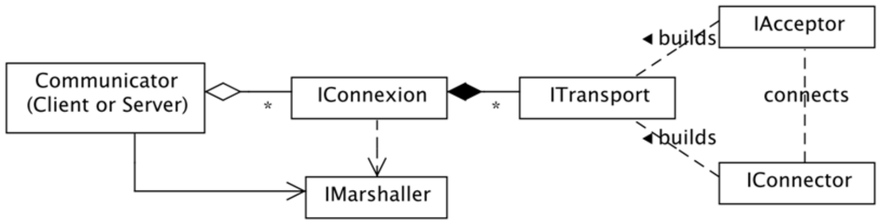

Figure 1: High-level design of the GT framework

3. OVERVIEW OF GT/SD

GT/SD is a layered toolkit for building and managing the networking and data sharing of real-time groupware applications.

The first layer is the Groupware Toolkit (GT), a modular communication framework supporting a typed messaging-oriented paradigm. GT handles many of the mundane aspects of network communication while still providing control over communication channels, especially those that could affect the perceived performance of groupware systems by end users.

The second layer is the Shared Dictionary (SD). It builds on top of GT to provide a distributed shared dictionary based on a publish/subscribe notification engine. This common data structure lets programmers think in terms of sharing data rather than networking. If greater control of network parameters is needed, SD give programmers access to certain features of the GT layer.

3.1 GT: The Groupware Toolkit

GT is a toolkit (currently written in C#) that provides message-based network connections between distributed computers. The core abstraction in GT is the *logical connexion* between two endpoints. Although the implementation has been primarily directed towards supporting client-server architectures, other higher-level architectures such as peer-to-peer architectures are also easily supported using this connection model.

GT exports a notion of connection-oriented, message-based communication between two endpoints. A logical connexion is divided into a set of one or more *channels*, which are allocated by the programmer to match the application's needs. Channels transport typed messages containing strings, byte arrays, objects, session notices, and typed 1-, 2-, and 3-tuples.

GT's modular design separates the different concerns involved in network communication (Figure 1). At a high level, programmers interact primarily with two classes representing *client and server communicators* (Figure 1, left). These two classes provide the bulk of the programmer-facing APIs. Client and server instances send and receive data with other remote endpoints through *connexions*. GT supports simultaneous use of a number of communication protocols, and each established network connection is represented as a *transport*. Thus, a connexion is a logical grouping of the different transports that connect to the same logical endpoint. A connexion uses a *marshaller* to transform a message to and from a byte array, and selects a transport for sending a message that best meets the message's delivery requirements. The *acceptor/connector* design pattern is used to separate establishing a transport from the actual delivery of packets [23]. Programmers can also add new behaviours by adding, wrapping, or replacing these well-defined components (discussed in more detail later).

GT provides four transports. First, a TCP-based transport provides *reliable and ordered delivery*. Second, a UDP-based transport provides unreliable and unordered delivery. Third, a sequenced UDP transport provides *unreliable but sequenced delivery*. Fourth, a local transport provides *reliable and ordered intra-process delivery* using a shared queue, which is useful for testing.

GT applications use a client-server architecture, where the server typically acts as a message repeater. This pattern has proven itself in networked games over the past decade [6] as well as in complex groupware systems [10]. The core idea is to centralize message-passing to reduce communication overhead between clients and guarantee all clients receive messages in the same sequence [9]. All messages are sent to a central server, which then broadcasts (or repeats) messages to all connected clients. We reified this pattern as the *ClientRepeater server* shipped with GT. This server can be extended, and thus acts as a starting point for applications requiring more server sophistication – for example, a server that processes incoming messages, or one that makes decisions about whether to relay messages to a particular client.

3.2 GT Example

Using the above facilities, a groupware programmer can easily create a networked groupware system. If and when performance needs warrant, the connection can be tuned to best fit the data needs sent over the connexion, and to choose the most appropriate transports that fit that data.

To demonstrate the simplicity of using the GT layer, we show the steps necessary to construct the simple chat client shown in Figure 2 (we deliberately use trivial examples in this paper to ease the reader's task of going through our code examples). Our chat client broadcasts messages to other connected clients via the client-repeater server. Because the server broadcasts messages to all clients including the one that sent the message, the chat client can respond to its own sent message in exactly the same way that other clients do. We do not show the actual UI code (which in fact dwarfs the GT networking code).

When the user presses <enter> after writing a new chat message in the lower text box (named messageBox in Figure 2), the client will send this message to the client-repeater. Clients receiving this message then append the text in the upper text box (named transcriptBox) to form a chat transcript.

First, we create and start a GT Client instance; this is often performed from the application constructor:

```
Client client = new Client();     // create a client instance
```

Next, we want a channel suitable for sending and receiving chat messages. We will use a channel supported by GT specifically for sending string objects. We assume that we already have the host and port IDs for the client-repeater, and that we have a unique channelNumber that identifies a new channel. As we do not want

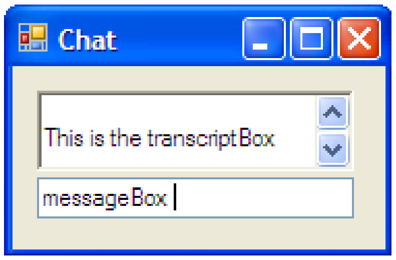

Figure 2: The Client Chat interface

to lose any chat messages, we use a predefined channel delivery method called ChatLike (an ordered and reliable method described in more detail later).

```
IStringStream chatChannel;
chatChannel = client.OpenStringChannel(host, port,
    channelNumber, ChannelDeliveryRequirementgs.ChatLike);
```

The network setup is now complete. To determine when to send text, we hook the messageBox's KeyPressed event. If the key is Enter, we send the messageBox text over the channel.

```
void messageBox_KeyPressed (object sender, KeyEventArgs e) {
    if (e.KeyCode == Keys.Enter) {
      chatChannel.Send (messageBox.Text);
      messageBox.Text = "" ;      }}
```

To receive messages, the application polls the incoming chat channel for messages, using a timer-invoked method. When messages are available, we update the transcript:

```
void timer_Tick(object sender, EventArgs e) {
    string msg;
    client.Update();           // process all connexions
    while((msg=chatChannel.DequeueMessage(0))!=null){
        transcriptBox.Text += msg + "\n";        }}
```

This example illustrates how easy it is to create a network connection in GT, but also hints at the powerful ways this connection can be configured.

3.3 SD: The Shared Dictionary

GT allows programmers to think about their systems in terms of *data networking* – what and how to send data over a channel. In many cases, however, it is far easier for a programmer to think about *data sharing* between clients – how data structures store information for shared access – rather than how that data should be delivered. Consequently, we developed the Shared Dictionary (SD) to provide a distributed shared-data abstraction.

The Shared Dictionary is a centrally coordinated shared-memory system for inter-application communication. It grew out of early work with shared dictionaries in GroupKit [22] and TeamRooms [10] (which they called 'environments'), and has been replicated as a distributed hash table in other systems (e.g., Sync [18]). The primary programming abstraction of SD is a distributed hash table, instantiated as a shared dictionary object. It contains hierarchical keys represented as string paths (e.g., /person/1/name, /person/1/age, etc.). Values can be any data or object, including multimedia. Programmers simply set data within the hash table, for example SD["/person/1/name"] = "Sam". A programmer can subscribe to notifications on any changes made to keys as specified by wildcards. For example, subscription.Pattern = "/person/*/name" will generate a notification whenever any person's name changes. Notifications typically trigger programmer-specified callbacks so they can act on these notifications. Each participant in an SD is identified by a globally-unique identifier that can be used as a component of an SD path.

As a shared data structure, all keys, values, and notifications are transparently shared across the distributed shared dictionary. Under the covers, clients connect and send data to a central server. Unlike GT's ClientRepeater, this server stores the data in its internal hash table, and may also persist the data for resiliency. The server also tracks all clients subscriptions, and forwards data changes only to the interested clients. Data is stored centrally so that latecomers can be brought up to date with other clients. None of the networking or architectural setup is normally seen by the groupware programmer, except for a few lines of code where the programmer indicates the location of the SD server.

SD's design also incorporates programming abstractions that are not only convenient for the programmer, but efficient in terms of performance. For example, sequential lists can be a nuisance to set up in a dictionary. SD provides a special object called a sequenced vector, where a programmer can set items in the vector by its absolute or relative position (e.g., SD["/item#3"] accesses the 4th item in the 0-based list, and SD["/item#-3"] accesses the 3rd item from the end). For efficiency, SD sends only the changed list items rather than sending the entire list.

SD supports several common patterns in groupware development. First, and most basic, it behaves as *shared memory*. If a programmer subscribes to all data (by subscribing to "*"), then they can access any data posted by others. Second, it can behave as a pure *notification server* [7][19], where data is propagated via notifications and handled purely by associated callbacks. Third, automatic notification combined with the ability to retrieve data at any time supports a *distributed Model-View-Controller* (dMVC) paradigm [11][22]. That is, a program is structured as a set of *controllers* that trigger data changes to the underlying model (typically in response to user input), *views* that respond to notifications about model changes (typically by adjusting what is seen on the screen), and the *model* that contains the state of the system (which can be accessed at any time to regenerate the view, e.g., by latecomers).

SD is built on top of the GT connection facilities. It uses GT for network communication, but abstracts this communication away from the programmer through the shared data structure paradigm. It automatically maintains the data model, and handles the client/server connections, data serialization, and data marshalling/unmarshalling.

3.4 Shared Dictionary Example

The code below shows the same chat client in Figure 2, but written as a distributed MVC pattern built atop SD. First, an SD-based chat client connects to the server via a shared dictionary object. We assume that the server has been started, and that its host/port ids are known.

```
SharedDictionary SD = new SharedDictionary();
SD.Url = "tcp://" + host + ":" + port;
SD.Open();
```

Second, we create a subscription to the key '/chat/message', a sequenced vector whose value will contain the contents of the entire chat transcript, i.e., the *model*. Note that when the value associated with this key is changed, our notification method chat_UpdateView (the *view*) will be called.

```
SharedDictionary.Subscription messageChanges =
                         new SharedDictionary.Subscription();
messageChanges.Dictionary = SD;
messageChanges.Pattern = "/chat/message";
messageChanges.Notified += chat_UpdateView;
```

When the user has finished composing a new message, we append it to the transcript value and thus update the model (i.e., the *controller*).

```
void messageBox_KeyPressed(object s, KeyEventArgs e) {
   if(e.KeyCode == Keys.Enter) {
      SD["/chat/message#-1"] = messageBox.Text + "\n";
      messageBox.Text = "";        }}
```

When notified, we add the new chat message to the transcript (i.e., the *view* is updated):

```
void chat_UpdateView(object s, SubscriptionEventArgs e) {
   transcriptBox.Text += e.Value;      }
```

If a new client arrives, they can immediately update their view simply by using the data values associated with the '/chat/message' key.

While the SD code above is slightly longer than the equivalent GT code, the differences concern simple boiler-plate one-time setup of the shared dictionary and subscriptions. At this point, it would be easy to create the quite complex shared data structures typically found in real groupware applications.

4. PERFORMANCE AND DEVELOPMENT

The above examples show that building basic groupware is simple using GT/SD. In the following sections, we detail the features of GT/SD that provide support for real-world performance and real-world development. In some of these features, programmers can set parameters to exploit knowledge of the data or the network; in other cases, the toolkit will automatically deal with performance issues. We begin with features that improve network performance, and then discuss development support.

4.1 Support for Latency Reduction

End-to-end network delay (latency) is a major problem for synchronous groupware. Since visual information about other people (e.g., the location of their avatar or telepointer) is vital in many groupware tasks, latency makes it difficult for people to coordinate shared actions or use deictic references [6]. Previous work shows that close coordination is disrupted at latencies above 100ms, and is nearly impossible at delays above 500ms [14].

Reducing latency is therefore a critical issue for the groupware developer. Previous work suggests that one of the most common problems is that applications exceed their available network bandwidth and fill up their own communication channel. This behaviour is particularly evident in systems built with earlier groupware toolkits such as GroupKit [22]. Fortunately, this problem can often be addressed by controlling both the amount of data sent and *how* that data is sent across the network. GT/SD provides four mechanisms for controlling outgoing data: rate control, aggregation, protocol choice, and compression.

Rate control. GT/SD provides two types of rate control. First, communication channels that are attached to specific local data values (e.g., the streamed tuple channel) allow the programmer to set specific send rates in milliseconds. Second, for channels where the system does not know what data will be sent next (e.g., generic text or object channels), GT/SD provides a standard programming pattern in which all message sending is put under the control of a send timer. Although not explicitly part of the toolkit, the tutorials and example programs highlight the need for rate control and the way it can be implemented.

Aggregation. Messages sent by groupware systems are often small (e.g., position updates typically only require a few bytes), and are generally much smaller than the maximum payload size of a network packet. Sending each message immediately in a single packet (often done in previous toolkits) wastes the space needed for extra packet headers, and wastes resources needed to process packets en route. Aggregation solves this problem by placing multiple messages into a single packet. Aggregation works by filling packets from an outgoing send queue: messages are aggregated until either the maximum packet size is reached, the queue empties, or a signal is received from the send timer. Although aggregation slightly increases the latency of individual messages, the savings in data volume can dramatically improve the overall delay.

Multiple reliability options. Several distributed-systems toolkits provide only reliable and ordered transmission using TCP (e.g., GroupKit, JSDT, Java RMI, C# Remoting). However, most messages in many groupware systems are awareness messages with no reliability requirements [6]. As a result, one of the main causes of latency in these systems is the unnecessary TCP-level retransmission of lost packets. For example, telepointer position updates are very small packets sent at a high data rate. It matters little if an update or two are lost, as the next one will contain sufficient information to update a client's telepointer position. In this situation, loss is preferable to delay.

An important part of reducing unnecessary data, therefore, is in providing appropriate reliability levels for messages, and in particular, providing unreliable transport that does not retransmit packets. GT/SD provides this flexibility with custom transport mechanisms built on top of the UDP and TCP protocols. Provision of unreliable transport is standard in game networking libraries, but is still uncommon in groupware toolkits.

Compression. Compression is a standard technique for reducing the size of digital data, and several game networking libraries provide mechanisms for message compression [6]. However, these mechanisms are either complex (e.g., requiring the application programmer to determine the minimum encoding required for the data ranges in the message [6]), or do not work well with the short messages that are common in real-time groupware (e.g., ZIP compression is not very effective on messages that are only a few dozen bytes long).

GT provides a message compressor – GMC – that provides good compression and requires very little programmer effort [13]. The key observation underlying GMC is that streaming groupware messages are often self-similar: because many of the fields in a message are repeated (e.g., tag names or participant IDs) one message in the stream is often very similar to the preceding and succeeding messages. GMC uses the messages themselves as templates, and replaces repeated sections of later messages with pointers to the template. Our experiments show that GMC reduces the size of simple text-based telepointer messages by more than 50% and works much better than message-at-a-time compression techniques such as ZIP [13].

Data compression is a good example of how knowledge of the specific behaviours and requirements of real-time groupware can lead to improved performance without sacrificing simplicity. A specific advantage of GMC is that it allows application programmers to use long message formats that contain redundant information (e.g., sending a participant's name as well as their numerical ID in each message). These long formats are useful for readability and debugging. With GMC, the redundant information is automatically removed by the compressor, ensuring that network performance is not compromised.

4.2 Application-Level Network / QoS Control

Many groupware applications require differing delivery requirements for particular types of data. For example, telepointer update messages have very different requirements compared with chat messages or model updates. Applications typically managed these requirements by using multiple communication protocols, such as using TCP for sending some data types and UDP for others. However, implicitly tying delivery requirements to particular protocols leads to brittleness, as substantial work may be necessary to adapt the application to a new, better-suited protocol.

GT uses a more robust approach. It allows programmers to explicitly specify the delivery requirements to be used for messages sent on a particular channel. Under the covers, these delivery requirements are used to select the most appropriate transport for sending the message. When needed, the programmer can override these requirements on a message-by-message basis.

GT currently supports the following four types of delivery requirements for each channel.

- *Reliability* describes the delivery guarantee for messages sent on a channel: is it required that the messages be delivered, or can they be sent on a best-effort basis?
- *Ordering* considers the delivery of a message with respect to the other messages on the same channel. There are three possibilities: unordered (delivery order does not matter); sequenced (messages received out of order should be dropped); and ordered (messages must be delivered in the order received).
- *Timeliness* describes the expected timeliness of the message: can a new message be held back to be aggregated with other messages? If not, should this message force sending all queued messages on this channel, on all channels, or must it be sent immediately before all other messages?
- *Freshness*. For channels configured to support aggregation, the channel can also be configured to specify the freshness of the messages sent on its channel: that is, whether all pending messages or only the latest message should be sent.

Ready-made defaults are available to simplify programming, tuned to the specific types of messages that are common in real-time groupware [6]. We have already shown the ChatLike default (reliable, ordered, aggregate messages, send all) in our GT Chat example. This ChatLike delivery requirement ensures that all messages are received in the order sent, but that brief delays are acceptable in order to aggregate messages. Other defaults include AwarenessLike (unreliable, sequenced, aggregate, latest-only), CommandsLike to represent user commands or model updates (reliable, ordered, flush-channel, all), SessionLike (reliable, unordered, flush-all, send all), and Data (reliable, ordered, aggregate messages, send all).

By default, SD assumes that all values require reliable and ordered delivery. However, programmers can attach meta-data attributes to key paths as hints to the underlying GT layer to change quality of service. For example, one attribute that can be attached to a path is 'unreliable', indicating a preference for data updates to be sent on a best-effort basis. This makes sense for many applications, such as a media space that sends video frames as sequential data updates: losing the occasional frame (which will likely be unnoticed by the end user) is preferable to flooding the network and introducing lag.

In summary, selecting a protocol by matching its delivery characteristics to the required delivery requirements reduces contention for bandwidth for other protocols, and thus helps the programmer manage network congestion.

4.3 Managing Different Types of Message Content

Groupware applications are fundamentally concerned with sharing information between different nodes. However, the actual form of the data can affect program complexity and/or how it is sent across the network.

Basic Data Types. As with most other network services, GT supports sending and receiving of primitive information data types, such as strings and byte arrays.

Objects. Simple data types often do not suffice. Most applications actually exchange objects, such as telepointer positions or objects containing data models. Forcing the developer to transform these objects into byte arrays or strings is onerous; instead, GT/SD provides direct support for sending application-level objects. A customizable marshaller is used to convert these objects to and from portable byte-based formats suitable for network transport. GT/SD's default marshaller uses .NET serialization, but other marshalling schemes are easily supported.

Large Objects. GT also provides a special marshaller for handling situations where the resulting byte format is too large for the underlying network's capacity. For example, UDP datagrams have a theoretical limit of 65536 bytes, although some operating systems limit this even further to 8192 bytes. Groupware programmers will often run into this limit when sending common data such as pictures or video frames. We solve this problem with a large-object marshaller that automatically splits objects into appropriately sized packets. Thus, groupware applications that send large objects can be built without extra coding.

4.4 Multiple Messaging Paradigms

When a message or object is received over the network, an issue that developers must deal with is how the message is presented to the program. Ideally, the method used will fit well within the programming paradigm used in the system. It is thus the responsibility of the groupware toolkit to communicate the message, object or updated value to the application in an appropriate way. GT/SD supports a combination of the approaches: polling, event-driven, data storage, and publish/subscribe. These correspond to the well-known push, pull, and lazy-update paradigms for sharing information. Since each of these approaches are appropriate in different groupware situations, GT/SD does not demand any particular paradigm.

GT Polling. The first method is polling (used in the GT chat example above). The application regularly checks to see if a channel has available content; if any exists, it is retrieved.

GT Events. The second method is event-based, where the toolkit automatically triggers an event when a message is received on the channel. This invokes a callback specified by the programmer. If our GT chat example used an event-driven interface, we would add an event listener when creating the chat stream. The method chats_MsgReceived would then perform the message dequeueing seen previously. No timer is needed:

```
chatChannel = client.OpenStringChannel(host, port, …);
chatChannel.MessagesReceived += msgsReceived;
```

SD Data Storage. The third method stores the value or object received over the network by updating the data structure held by the Shared Dictionary. These values are retrieved by accessing the data structure directly.

SD Publish/Subscribe. Finally, the publish/subscribe paradigm allows the programmer to selectively subscribe to data patterns held by the shared dictionary. This is valuable for several reasons. First, unsubscribed data is not sent to that particular client, which relieves both network and memory load. Second, subscriptions offer a very easy way to implement a notification server and/or the dMVC pattern. Publish methods are equivalent to a controller, while subscribe callbacks are usually equivalent to the view.

These mechanisms illustrate the design goal of providing flexibility where it is needed. GT's core functionality provides two perspectives on providing multiple message paradigms, each oriented towards building either the server side or client side of a groupware application. Our server-side perspective is entirely event-driven, because servers generally concentrate on minimizing the response latency between a message being received, processed, and the dispatching of any result arising from the message. On the other hand, our client-side perspective offers a combination of approaches using event handlers or polling. Similarly, SD's server uses somewhat more intricate methods to decide what messages to forward to clients (i.e., corresponding to subscriptions). However, its clients can use either data storage, or publish/subscribe, or both.

4.5 Debugging Support

Debugging a distributed system such as a groupware application is difficult because the execution state is defined by the state of multiple nodes. Recreating an erroneous state may depend on specific orderings of how messages were received and processed by the individual nodes. Any slight perturbation in the communication patterns, such as might occur when interrupting a node with a breakpoint, may mask the bug entirely (that is, a *heisenbug*). Even when the erroneous state can be recreated, it is often only possible to approach the situation from a post-mortem perspective: it is normally impossible to coordinate a simultaneous suspension of the other nodes, where the remainder of the system will continue to execute with attendant changes to communication patterns.

Pedersen and Wagner studied debugging parallel programs in their in Millipede system [20]. Their key insight was that certain classes of parallel or distributed systems can be reduced to debugging sequential programs. The proviso is that the program executed by a node is deterministic in response to its incoming network traffic. In such situations, a program can be debugged as an independent process simply by replaying the received messages.

GT/SD supports this Millipede-style packet recording and replay, and requires no changes in the applications. If the programmer wants this to occur, he or she merely sets a special debug environment variable. On startup, GT client and server processes check this variable for one of three values: "record" to record the incoming and outgoing messages for this session to a file, "replay" to replay the messages from a file, and "passthrough" to run as normal. When replaying, the program runs in complete isolation from any GT-related communication, and no GT-related traffic is sent live to the network. Since the program runs in isolation, developers can use their traditional tools for debugging sequential programs, such as using breakpoints for *in vivo* examination of their system.

Our debugging support may not exactly reproduce the recorded behaviour when programs are non-deterministic for reasons other than GT communication (e.g., keyboard and mouse input or timeouts). There are, however, workarounds for these cases, such as integrating the sources of the non-determinism into the GT-Millipede framework.

4.6 Testing Support

Testing and monitoring of applications is done via various GT/SD tools. First, GT includes a local transport that channels all communication through a shared queue; this transport is helpful for isolating networking problems, for creating unit tests, as well as for GT-Millipede replay. Second, GT includes a statistical monitoring package. It produces graphs of various statistics, such as the numbers of messages and packets both sent and received, average round-trip latency, and per-protocol statistics. Under the covers, GT is self-instrumenting, where it monitors activity through its events mechanism. Third, SD includes several tools for monitoring and manipulating the contents of a shared dictionary, and for performing speed tests. Programmers (and end users) can optionally see the contents of the shared dictionary as it is being updated, and can even add, delete, or change values through an external tool to see what effects it will have.

4.7 Extensibility and Experimentation

We wanted GT/SD to be extensible for two reasons. First, we know that we will want to add new features to it as we gain experience developing groupware, especially applications that are quite different from those we are now working on. Second, we are researchers, and we want GT/SD to serve as an experimental platform that will allow us to prototype and test new toolkit features.

GT's modular design permits adding or modifying behaviour by replacing or wrapping different components. Indeed, we have implemented significant portions of GT's functionality simply by wrapping existing components:

- the GMC message compressor is implemented as a wrapper around a marshaller;

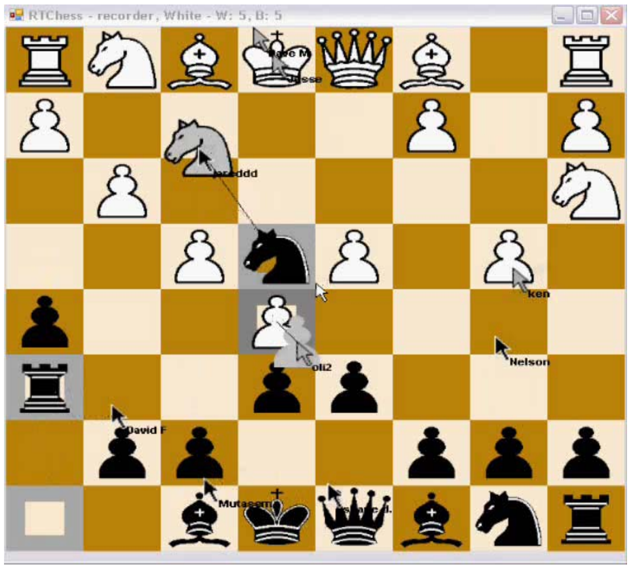

Figure 3: RTChess, built with GT. Ten players are represented with telepointers and name tags. Players jareddd and oli2 are moving pieces; the black rook has just been moved.

- the GT-Millipede functionality is implemented by wrapping transport, acceptor, and connector objects;
- we have implemented variants of the leaky-bucket and token-bucket traffic-shaping algorithms [27] as a wrapper around a transport;
- we have implemented a custom marshaller for one application to produce minimal-sized messages, with no attendant effects on the application;
- the ability to wrap components helped us investigate the effects of delay on group performance [26].

GT also exports a variety of fine-grained events to provide notification of network events. These include the creation and removal of connexions and transports, of messages being received and sent, of ping round-trip messages. We have used these events to provide diagnostic functions, and to provide application-level disconnects such as when a repeated number of pings fail.

5. EXPERIENCES WITH GT/SD

GT/SD stakes out the middle ground of groupware toolkits. For those building prototypes, its basic networking and shared data facilities likely suffice: simple things are kept simple. For those who need to build robust Internet applications, it provides control over vital aspects of groupware networking, debugging and experimentation with only a modest amount of extra effort by the programmer.

Our earlier example of a chat system shows how simple GT and SD application development can be. However, the toolkit can be used for much more complex systems. Using GT, we have built several group games, a distributed chalkboard application, and a screen-sharing system. Similarly, SD has been used for many different applications – for example, the powerful Shared Phidgets project that uses dMVC to collect, store and propagate values between hardware devices, as well as visualize sensor and actuator traffic between them [17].

The examples below illustrate other projects in somewhat more detail. While they are not an empirical proof that GT/SD is 'easier' to program than other systems, or that it offers better performance, we do claim that our experiences have been positive in both regards.

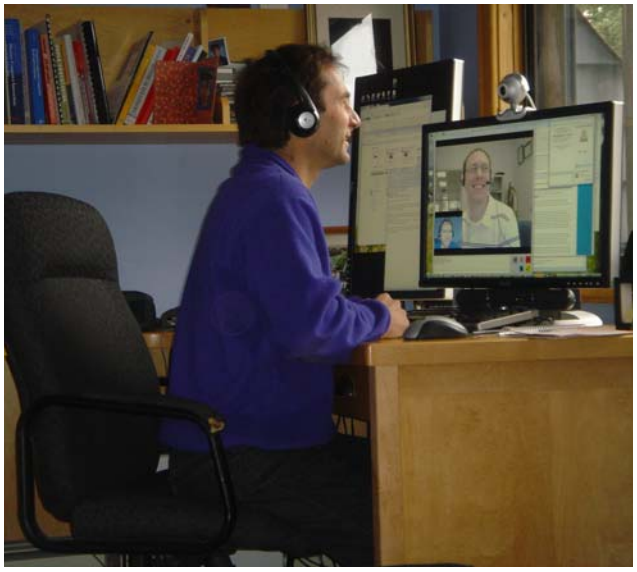

Figure 4: ME-dia space home node, built using the SD. The distant person is visible in the large video, while the local person's sent video is mirrored at the lower left to provide feedback.

GT Example: Real-Time Chess. As a more concrete example, Real-Time Chess (Figure 3) is a fast-paced multiplayer game based on the classic board game [12]. While most of the rules of chess still apply, any number of players can join, any player can move any of their team's pieces, a king can move into check, and movement of pieces is limited only by the players' speed in manipulating the interface and the network delay to the server. The result is a very fast chess variant.

RTChess was built using two GT channels. The first channel carries awareness and presence information, such as telepointers, and is configured with AwarenessLike delivery requirements; this channel was updated every 0.1 seconds. The second channel carries game commands, such as join-game, select-piece, or move-piece, and is configured with Commands delivery requirements. We implemented an earliest-arrival server to arbitrate commands: clients make requests of the server, and the server either sends the resulting commands to the group, or sends refusals to individual clients (e.g., the piece requested is no longer in play).

To minimize network latency, we implemented a custom marshaller to send minimally-sized packets using 317 lines of code. Typical game messages are expressed as a dozen bytes or less, as compared to a minimum of 300 bytes when using the .NET Serialization-based marshaller. The marshalling knowledge is entirely contained within the RTChess marshaller, and could be substituted with the .NET Serialization marshaller with no effect on the rest of the application.

SD Example: The ME-dia Space. Media spaces contain an always-on video connection between people. We constructed the ME-dia Space [29], shown in Figure 4, which connects a telecommuter working at home with his work office. People would then drop into the work office to talk to him. It sends streaming video frames, sensor data (to signal when people enter the office), actuator data (to remotely open/close the work office door), and a text chat system (voice is handled via the Skype API). While it generally uses SD to share data, it also uses the GT layer to transmit video frames in a performance-efficient way.

6. DISCUSSION

In this section we discuss several questions that arise from our experiences with the GT/SD toolkit, including issues of generalization, flexibility, and design rationale for our approach.

Are performance and simplicity mutually exclusive?

Earlier we characterized previous toolkits in terms of a tradeoff between simplicity and power – but these two qualities are not necessarily an either-or proposition. Although the toolkits that inspired GT/SD have focused on either network performance (e.g., game libraries such as OpenTNL) or simplicity (e.g., GroupKit [22]), there is no real reason that systems cannot provide both. A toolkit should simplify at least some otherwise-complex operations: even game toolkits, which we characterize as 'powerful but difficult', can be seen as providing simpler APIs than the raw socket implementations that they are built on. Our goal in GT/SD is to provide important performance capabilities without increasing programmer effort to the point where researchers and students would be unable to quickly and easily build prototypes. We attempt to achieve this in three ways: first, by building certain capabilities in as automatic features (such as the SD layer, or GT's compression module) that improve performance without the application programmer having to do any extra work; second, by building the toolkit around patterns that avoid common design errors (e.g., the error of sending telepointers at a high rate through TCP [6]); and third, by abstracting the design in ways that are easier for groupware programmers to understand and optionally extend (e.g., providing QoS defaults such as 'chat-like' or 'awareness-like', rather than requiring knowledge of different network transports).

Why focus on the qualities of performance and simplicity?

Our main motivation in developing GT/SD was that groupware prototypes built from existing toolkits did not work well when deployed on real-world networks. Therefore, our goal was to provide a toolkit that would be as simple (or nearly so) as earlier examples, but that would allow researchers and developers to test groupware in realistic network situations. Groupware toolkits in general have tended to focus on a fairly narrow set of concerns – which is understandable, given the difficulty and effort needed to produce and maintain a robust and usable library. Therefore, we chose to focus on network performance and development simplicity at the expense of other possible features and qualities such as flexibility in architectural style, user-interface coupling, collaboration transparency, or consistency management. Additional features will be added in future (as described below), but we plan to maintain a relatively small set of core functionality that concentrates on our main goals.

In support of this approach, earlier reviews (e.g., [23]) suggest that utility layers and middleware should reduce their feature count, since toolkit designers are rarely able to anticipate the different needs faced by application programmers. Focusing on network performance may therefore be a more general approach, since most real-time groupware systems must deal with performance issues. In contrast, issues of architectural style or data consistency are often highly application dependent. Tools should be available for programmers to deal with these issues, but they can be dealt with through additions to the toolkit or separate modules.

What types of groupware are best suited to GT/SD?

The design of GT/SD means that the toolkit's strengths can be seen most easily in a particular class of groupware – that is, distributed client-server systems that send a mixture of message types (i.e., mostly awareness messages with some transactions). In this design we are following the lead of networked-game libraries, who have validated this architectural style over several years. GT on its own best supports groupware requiring efficient transfer of data, while SD works well with either dMVC or notification-based styles. The most common application types that fit these constraints include communication systems (e.g., instant messaging or awareness systems), systems with shared visual workspaces (e.g., drawing and design applications), and real-time multi-player games (e.g., real-time chess as described above). There are limits in the current version of GT/SD, however, and so not all applications in each of these genres are suitable for the toolkit. First, GT/SD does not provide streaming video or audio transmission, so real-time conferencing must be handled out of band. Ideally, we would like to add another facility to GT called 'stream-like' which could support this type of data. Second, GT does not yet supply a consistency-management system, which means that shared editors built with GT/SD must have only minor data dependencies (in our experience, however, the 'social locking protocols' that are easy to implement through awareness support provide sufficient consistency maintenance for most shared-workspace systems [9]). Third, GT does not provide the types of game tools (e.g., 3D support, level builders, cheating detectors) that are seen in other gaming libraries, so the games built with GT/SD are more likely to feature simpler graphics and are more likely to be explorations of new game ideas (such as real-time chess) rather than production systems. Although GT/SD is primarily designed for real-time and distributed groupware, however, it can be used to build asynchronous systems, co-located groupware, and intermediate forms between these endpoints. For example, we are using the toolkit to connect two tabletop groupware systems, making a hybrid of co-located and distributed interaction.

7. CONCLUSIONS AND FUTURE WORK

We have introduced the GT/SD toolkit, a new toolkit that provides good real-world network performance while maintaining programming and development simplicity. The base GT layer provides core networking and development services, and the SD layer builds onto GT to provide a simple yet powerful programming abstraction for rapid prototyping. GT/SD implements ideas from previous groupware toolkits and from game libraries that help solve problems of network delay, quality of service, rapid development, flexibility, and testing.

Our experiences with the toolkit suggest that we have succeeded in meeting our design goal – that it should be easy to build groupware that performs well on real-world networks. GT/SD dramatically simplifies network setup and data sharing, provides simple access to vital aspects of network control, and still gives full power to application developers when needed.

GT/SD solves many problems facing groupware developers, but does not address all issues – many additional groupware capabilities and features could be added to the toolkit. We plan two streams of further work. First, we will improve the core networking capabilities of GT/SD and add further application-level networking techniques, deeper QoS support, firewall traversal, and more comprehensive network monitoring.

Second, we will add several modules that provide other important groupware services. Included in this list are features such as application-level concurrency control [9], groupware widgets [15][22], real-time protocols for audio and video, persistence and disconnection support, data consistency and convergence techniques, and deployment of GT/SD to mobile platforms.

Acknowledgements

Several people have worked on versions of GT and SD, including Dane Stuckel, Mike Boyle, Chris Fedak, Mark Watson, and Adrian Reetz. We gratefully acknowledge support from NSERC and the NECTAR research network.

Software, documentation, and examples available at hci.usask.ca/research/gt/ & grouplab.cpsc.ucalgary.ca/cookbook/

REFERENCES

[1] Begole, J., Struble, C., Shaffer, C. & Smith, R. Transparent sharing of Java applets: A replicated approach. *Proc. ACM UIST*, 55-64, (1997)

[2] Chung, G. & Dewan, P. Towards dynamic collaboration architectures. *Proc. ACM CSCW*, (2004)

[3] Crowley, T., Milazzo, P., Baker, E. & Forsdick, H. MMConf: An infrastructure for building shared multimedia applications. *Proc. ACM CSCW*, (1990)

[4] Dewan, P. A tour of suite user interface software. *Proc. ACM UIST*, 57-65, (1990)

[5] Dourish, P. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM TOCHI*. 5(2), 109–155 (1998)

[6] Dyck, J., Gutwin, C., Graham, T.C.N. & Pinelle, D. Beyond the LAN: Techniques from network games for improving groupware performance. *Proc. ACM GROUP*, 291–300, (2007)

[7] Fitzpatrick, G., Kaplan, S., Mansfield, T., Arnold, D. & Segall, B. Supporting public availability and accessibility with Elvin: Experiences and reflections. *Comp. Supp. Coop. Work*, 11(3-4), 447–474, (2002)

[8] Greenberg, S. Toolkits and interface creativity. *J. Multimedia Tools and Applications*, 32(2), 139-159, (2007)

[9] Greenberg, S. & Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. *Proc. ACM CSCW*, 207-17, (1994)

[10] Greenberg, S. & Roseman, M. Using a Room Metaphor to Ease Transitions in Groupware. In *Sharing Expertise: Beyond Knowledge Management*. (M. Ackerman, V. Pipek, V. Wulf, Eds.) MIT Press, (2003)

[11] Graham, T. N., Urnes, T. & Nejabi, R. Efficient distributed implementation of semi-replicated synchronous groupware. *Proc. ACM UIST*, 1-10, (1996)

[12] Gutwin, C., Barjawi, M. & de Alwis, B. *Chess as a twitch game: RTChess is real-time multiplayer chess.* Demonstration at ACM CSCW, (2008). (Summary at hci.usask.ca/publications/2008/cscw-demo/summary.pdf)

[13] Gutwin, C., Fedak, C., Watson, M., Dyck, J. & Bell, T. Improving network efficiency in real-time groupware with General Message Compression. *Proc. ACM CSCW*, 119–128, (2006)

[14] Gutwin, C., Benford, S., Dyck, J., Fraser, M., Vaghi, I. & Greenhalgh, C. Revealing delay in Collaborative environments, *Proc. ACM CHI*, 503-510, (2004)

[15] Hill, J. & Gutwin, C. The MAUI toolkit: Groupware widgets for group awareness. *Comp. Supp. Coop. Work*, 13(5-6), 539-571, (2004)

[16] Kendall, S.C., Waldo, J., Wollrath, A. & Wyant, G. *A note on distributed computing*. Tech. Rep. TR-94-29, Sun Microsystems Inc., (1994)

[17] Marquardt, N. & Greenberg, S. Distributed physical interfaces with Shared Phidgets. *Proc. ACM Tangible and Embedded Interaction*, 13-20, (2007)

[18] Munson, J. & Dewan, P. Sync: A Java Framework for Mobile Collaborative Applications, *IEEE Computer*, 30(6):59-66, June (1997)

[19] Patterson, J.F., Day, M. & Kucan, J. Notification servers for synchronous groupware. *Proc. ACM CSCW*, 122–129, (1996)

[20] Pedersen, J.B. & Wagner, A. Sequential debugging of parallel message passing programs. *Proc. Commun. in Comput. (CIC)*, 55–61, (2000)

[21] Rodden, T., Mariani, J. A. & Blair, G. Supporting cooperative applications. *Comp. Supp. Coop. Work*, 1(1-2), 41–67, (1992)

[22] Roseman, M. & Greenberg, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM TOCHI*, 3(1), 66–106, (1996)

[23] Saltzer, J., Reed, D., and Clark, D. End-to-end arguments in system design. *ACM Trans Comput. Syst.* 2(4):277–288, (1984)

[24] Schmidt, D.C. Acceptor and connector: A family of object creational patterns for initializing communication services. *Pattern Languages of Program Design 3*. Addison-Wesley, (1997)

[25] Schneiderman, B. Creativity support tools: A grand challenge for HCI researchers. In *Engineering the User Interface* (M. Redondo, Ed.), Springer, (2009)

[26] Stuckel, D. & Gutwin, C. The effects of local lag on tightly-*coupled interaction in distributed groupware. Proc.* ACM CSCW, 447–456, (2008)

[27] Tanenbaum, A.S. *Computer Networks*. Prentice-Hall, 4th edition, (2003)

[28] Tse, E. & Greenberg, S. Rapidly prototyping single display groupware through the SDGToolkit. *Proc. 5th Australasian User Interface Conference*, Australian Computer Society Inc., 101-110, (2004)

[29] Voida, A., Voida, S., Greenberg, S. & He, H. Asymmetry in media spaces. *Proc. ACM CSCW*, 313—322, (2008)

[30] Wolfe, C., Smith, J., Phillips, W.G. & Graham, T.C.N. A model-based approach to engineering collaborative augmented reality, in *Engineering of Mixed Reality*, (E. Dubois, P. Gray & L. Nigay, Eds), Springer, (2009)