Combining Power and Simplicity in a Groupware Toolkit

Brian de Alwis[1], Carl Gutwin[1] and Saul Greenberg[2]
[1]University of Saskatchewan, Saskatoon, Canada
[2]University of Calgary, Calgary, Canada
[brian.de.alwis, Gutwin]@cs.usask.ca, saul.greenberg@ucalgary.ca

ABSTRACT
Many tools exist for the development of real-time distributed groupware, but most of these tools do not provide an appropriate balance of power and simplicity for necessary for prototyping or research. To better support the middle ground in groupware development, we built a new toolkit called GT/SD. It provides solutions to problems of real-world network performance without sacrificing the simple programming approach needed for rapid prototyping. GT/SD builds on the successes both of earlier groupware toolkits and game networking libraries, and implements seven ideas that help solve problems of rapid development, network delay, quality of service, and testing. We introduce the design and the benefits of GT/SD, and demonstrate the toolkit through several examples.

Keywords
Toolkits, network programming, groupware, extensibility

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 [18], JSDT [13], or Fiaa [25]. Lower level distributed-object protocols are being built into development environments (e.g., RMI for Java or Remoting for C#). 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) [5,20]. Very few tools do both. For example, game toolkits provide power, but are complex, have a steep learning curve, and are often difficult to use. Thus they are generally adopted only by those highly skilled developers committed to game production. At the other end, groupware toolkits like GroupKit [18] are easy to learn by average programmers. Yet because they hide almost all internal details, GroupKit applications may not perform well in real-world settings, or programmers may hit ceilings that stop their explorations [5,20]. Thus programmers are typically restricted to proof-of-concept prototypes.

The problem is that no existing groupware toolkit takes the middle ground of trying to provide control over certain vital aspects of groupware networking while maintaining a simple approach for the application programmer. Yet 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 intended for use across the Internet or mobile networks. Simplicity is also important, for rapid prototyping of new ideas may not warrant full-blown development effort, or for groupware researchers and students who create many different prototypes rather than focusing on a longer-term product line.

In response to this need for a more judicious combination of simplicity and power in groupware development, we built a new groupware toolkit called the *Groupware Toolkit/Shared Dictionary* (or GT/SD for short). GT/SD focuses on networking and data sharing. It grows out of several years of experience working with groupware tools, and is designed to support rapid development of groupware that can run successfully on real-world networks. GT/SD goes beyond basic connectivity to help solve several of the practical problems faced by developers of real-time, real-world groupware. In this paper, we focus on seven ideas that set GT/SD apart in the world of groupware toolkits:

- *Managing different types of message content*, from basic data types to custom objects to very large objects;
- *Multiple messaging paradigms* let developers choose amidst polling, events, shared data or publish/subscribe;
- *Latency support* provides techniques to avoid situations where groupware systems exceed available bandwidth (rate control, aggregation, reliability, compression);
- *Application-level network control* gives different service levels (reliability, ordering, timeliness, freshness) tuned to common groupware messaging requirements;
- *A generic shared dictionary* to allow rapid prototyping of data-intensive applications
- *Development support: debugging* through replay mechanisms, and *testing* via statistics gathering;
- *Extensibility* to enable experimentation and support novel research ideas.

The design philosophy underlying the GT/SD toolkit is:

Basic groupware should be trivial, good network performance should be only slightly more difficult, and full control should be possible for those that want it.

Cite as:
De Alwis, B., Gutwin, C. and Greenberg, S. (2009) Combining Power and Simplicity in a Groupware Toolkit. Report 2009-923-02, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4

That is, simple things should be simple and difficult things should be possible [5]. In the next sections we review previous research into groupware toolkits, provide a high-level overview of GT/SD, and then discuss each of these seven aspects of the toolkit.

RELATED WORK

GT/SD evolved through our experiences with three areas of previous work: other groupware toolkits, distributed-systems support in programming infrastructure, and game networking libraries. Much of this prior work can be organized at a high level by the idea of *distribution transparency* [16], that is, the degree to which the environment hides the details of the distributed system from the application programmer.

Distributed-systems infrastructure

Capabilities for building networked systems have now been built into many programming languages and environments. The sampling below includes sockets, RPC mechanisms, middleware, and full distributed-object systems.

Sockets. The BSD Sockets interface has become the *de facto* network programming interface. Sockets provide a very low-level, byte-oriented perspective on network communication. Although sockets provide maximum control, they are very tedious to program for all but the simplest networking applications. Thus network application programmers inevitably build their own (typically buggy) higher-level communication primitives (thus reinventing the wheel), or use some other communication library.

Remote Procedure Call (RPC) mechanisms. Java RMI and .NET Remoting are current examples of synchronous RPC systems. RPC systems disguise client/server network communication as method calls from a client to a server. While powerful, the RPC model breaks down when exposing and handling networking problems as they occur [13]. The model further breaks down when trying to represent data sharing situations that do not match RPC semantics. These RPC systems do not expose any part of their implementation.

Middleware. Loosely-coupled communication middleware solutions, such as store-and-forward messaging-queues (e.g., IBM MQseries, Java Message Service) offer guaranteed delivery, but sacrifice real-time or near real-time performance. These guarantees can be overkill, as there are many groupware exchanges where such guarantees are not required (e.g., telepointer updates).

Distributed-object systems. Industrial-strength approaches, such as CORBA and J2EE, are powerful but are far too heavy-weight for the need of most groupware developers. In particular, they require a substantial investment in knowledge to be used effectively.

Groupware toolkits

Previous groupware toolkits have focused on different 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., Clock [8])
- 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 [23]);
- simple development of groupware, including simple programming paradigms (e.g., Groupkit [18]) and provision of groupware widgets (e.g., MAUI [11]);
- investigation of metaphors for organizing groupware applications, such as a rooms environment [7].

These toolkits emphasize particular themes in developing groupware applications, but at the cost of being highly constrained. That is, they are specific solutions to particular groupware problems, rather than general solutions to problems of real-world groupware development. In addition, none of these toolkits provided techniques for good network performance over wide-area networks.

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. The network gaming industry has more than a decade of experience in delivering a high-quality multiplayer experience over the Internet, with millions of users and thousands of game titles. Such 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., [3], there are three main types of techniques:

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

Game libraries provide the largest number of techniques for tuning network performance, and the most control over the behaviour of the application. However, these libraries are not well suited to the needs of many groupware developers. While hard things are made possible by these libraries, simple groupware is difficult to build. First, game libraries are complex and difficult to learn, with

large APIs and unintuitive programming conventions. Second, game libraries require that application programmers understand concepts of underlying network behaviour, such as loss patterns, and implementation details such as Nagle's algorithm or socket buffer sizes. 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 types of groupware like shared editors, communication systems, or more casual games.

In summary, the different design goals of current generation distributed systems infrastructures, groupware toolkits, and game networking libraries tools may be well-suited to particular needs, but do not support the general needs of a groupware developers who are building groupware prototypes or research systems. Consequently, we developed the GT/SD toolkit to focus on simple development of groupware that still maintains good performance on the real-world Internet. As we will see, GT/SD synthesizes methods and lessons learnt both from previous groupware toolkits and from game libraries, combining them in ways that allow simplicity and power for many groupware applications.

GT/SD: KEEPING SIMPLE THINGS SIMPLE

GT/SD is a layered toolkit for building and managing the networking and data sharing necessary for groupware applications. The first layer is the Groupware Toolkit (GT), a modular communication framework supporting a typed messaging-oriented paradigm. GT handles much 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 of data based on a publish/subscribe notification engine. This common data structure lets programmers think in terms of sharing data rather than networking. However, SD has methods that the programmer can invoke to influence how the underlying GT layer works. We discuss each layer in turn below.

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 connection* 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 connection 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.

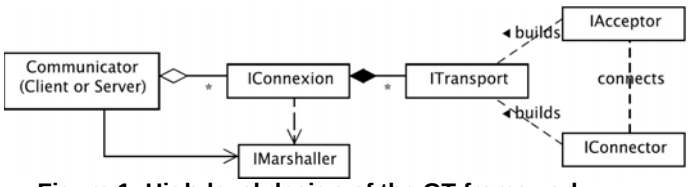

Figure 1: High-level design of the GT framework

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 side). 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 [19]. Programmers can also add new behaviours by adding, wrapping, or replacing these well-defined components with no impact to the application. For example, we have used wrapping as a technique for creating a compressing marshaller, and for adding traffic shaping restrictions to an existing transport (discussed in more detail later).

As mentioned, GT supports communication over multiple transports with differing transport characteristics. 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 purposes.

GT applications use a client-server architecture, where the server acts as a message repeater. This pattern has proven itself in networked games over the past decade [3]. The core idea is to centralize message-passing to reduce communication overhead between clients. 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. Thus it acts as a feasible starting point for those applications requiring more server sophistication, e.g., one that actually processes incoming messages, or one that makes decisions on whether to broadcast a message to a particular client.

GT Example

Using the above facilities, a groupware programmer can easily create a networked groupware system. If and when

performance needs warrant it, 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.

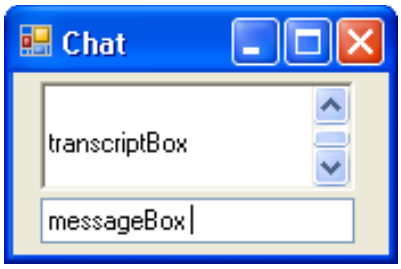

Figure 2: The Client Chat interface

To demonstrate the simplicity of using the GT layer, we show the steps necessary to construct the simple chat client shown in Figure 2. 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. Due to space concerns, we do not show the actual UI code (which in fact dwarfs the GT networking code).

When the user hits enter after composing a new chat message in the lower text box (named messageBox), the chat 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 UI Form 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/port ids for the client-repeater, and that we have a unique channelNumber that identifies a new channel. We don't want to lose any chat messages, so we use a predefined channel delivery method called ChatLike (an ordered and reliable method described in more detail later).

```
IStringStream chatChannel;
chatChannel = client.GetStringStream(host, port,
            channelNumber,
            ChannelDeliveryRequiremengts.ChatLike);
```

The network setup is complete. To send text, we invoke the KeyPressed event of the messageBox. 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, we use polling (an event-driven approach is also possible). In this example, we poll the chat channel for messages from 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. In spite of this simplicity, it hints at the powerful ways this connection can be configured.

SD: The Shared Dictionary

GT works when programmers are thinking in terms of networking. Yet in many cases, it is much easier for a programmer to think about the *data* that is shared between groupware process instances, rather than in how that data should be delivered over the network. Consequently, we developed the Shared Dictionary (SD) to let programmers work within a distributed shared data abstraction.

The Shared Dictionary is a centrally coordinated shared-memory system for inter-application communication. In essence, 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 wildcarding. 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. To avoid namespace collisions, each participant process in an SD is identified by a globally-unique identifier, or GUID, and GUIDs are used to differentiate data associated with one client from data associated with another client. That is, they can be used anytime as a component of a path for accessing and storing data specific to a particular client using the SD.

What is important is that all keys, values, and notifications are shared across all distributed groupware processes that use a shared dictionary instance. 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. The server also tracks all clients that have subscribed to that data, and forwards a copy of changed data *only* to those clients. Storage is done so that latecomers can request a copy of the data, and thus come 'up to date' with other processes that comprise the groupware system. All networking activity, communication, and architectural setup is not normally seen by the groupware programmer, excepting a few lines of code where the programmer indicates the location of the SD server.

SD caters to several common patterns in groupware development. First, and perhaps 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 directly. Second, it can behave as a pure *notification server* [15,4], where data propagated throughout the system is monitored via notifications and handled via associated callbacks. Third, automatic notification combined with the ability to retrieve data at any time supports a *distributed Model-View-Controller* (dMVC) paradigm. That is, a program is structured as *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 layered 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, handles the client/server connections, data serialization, and data marshalling/unmarshalling.

Shared Dictionary Example

The code below shows the same chat client in Figure 2, but written using SD using a distributed MVC pattern.

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', 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 change the transcript value and thus update the model (i.e., the *controller*).

```
void messageBox_KeyPressed(object sender,
                                    KeyEventArgs e) {
   if(e.KeyCode == Keys.Enter) {
      SD["/chat/message"] += 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 sender,
                         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 a bit longer than the equivalent GT code, most of it involves simple boiler-plate one-time setup of the shared dictionary and subscriptions. At this point, it would be easy to create quite complex shared data structures typical in real groupware applications.

GT/SD: MAKING DIFFICULT THINGS POSSIBLE

The above examples and descriptions show that simple things are indeed simple with GT/SD. We now describe in somewhat more detail how GT/SD makes harder things possible, where it addresses the practical problems faced by developers of real-time distributed groupware. The various points below reveal the more sophisticated aspects of the GT/SD toolkit. In some cases, programmers can optionally set toolkit parameters to exploit their knowledge of the data being transmitted or network performance bottlenecks. In other cases, the toolkit itself will automatically deal with these aspects under the covers.

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. Along 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 telepointers or those containing large data models. Forcing the application developer to transform these objects into primitive bytes 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 with no negative impact on application programming.

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 manages object splitting into appropriately sized packets. Thus groupware applications that send large objects can be built without extra coding.

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 sys-

tem. 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. Each approach involves trade-offs, which is why GT/SD does not demand any particular approach. Rather, it lets the programmer choose the approach that best fits their programming paradigm.

GT Polling. The first method is polling (as illustrated in the GT chat example above). The application regularly checks the channel to see if it has any content. If content exists, it pulls that content off the channel.

GT Events. The second method is event-based, where the toolkit automatically triggers an event when a message is received over the channel. This in turn invokes a callback specified by the programmer. For example, in redoing our GT chat example as 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.GetStringStream(host, port, …);
chatChannel.msgsReceived += chats_MsgReceived;
```

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.

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 communicate through deictic references [3]. Previous research shows that close coordination begins to be disrupted at latencies above about 100ms, and becomes nearly impossible at delays above 500ms [11].

Reducing latency is therefore a critical issue for the groupware developer. However, it is not always clear what can be done. Previous work suggests that there are several sources of delay, but 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 [18]. Fortunately, this problem can often be addressed by controlling the amount of data that a groupware system sends 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 require only 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 [3]. 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 con-

tain 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 [3]. 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 [3]), 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 [10]. 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 [10].

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 extremely useful for readability and debugging. With GMC, the redundant information is automatically removed by the compressor, ensuring that network performance is not compromised.

Application-Level Network & Quality of Service 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.

- *Channel reliability* describes the delivery guarantee of messages sent on this channel: is it required that the messages be received or can they be sent on a best-effort basis?
- *Ordering* describes the required ordering of a new message with respect to the other messages sent on the same channel. There are three possibilities: unordered, such that the order received does not matter; sequenced, such that messages received out of order should be dropped; and ordered, where messages must be presented 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, specifically whether all messages sent on the channel must be sent, 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 [3]. 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 TelepointerLike (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, the programmer can optionally attach meta-data attributes to key paths as hints to the underlying GT layer to change its quality of service methods. For example, one attribute that can be attached to a path is 'unreliable', where the programmer is stating a preference for data updates to be sent on a best-effort basis (e.g., through a UDP-like channel). This makes sense. For example, consider a groupware media space that sends web cam video frames as sequential data updates: losing the occasional frame (which will likely be unnoticed by

the end user) is vastly preferred 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.

Development 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 & Wagner researched debugging parallel program in their in Millipede system [16]. 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.

This debugging support may not reproduce the recorded behaviour when the program is non-deterministic for reasons other than its GT communication. Such non-deterministic behaviour can arise for a variety of reasons, such as dependencies on changed external inputs (e.g., keyboard and mouse input, or non-GT communications), or time-based assumptions such as timeouts or delays. There are several possible workarounds to avoid these non-determinisms. One approach is to have all user input processed by a server, such that the responses from the server will be recorded and subsequently replayed with no need for user interaction. Another approach is to specify parameters through the command line, or automating the program behaviour in the face of Millipede recording/replaying. Alternatively, these sources of non-determinism can be integrated into the GT-Millipede framework to record and replay the values seen, but such details are beyond the scope of this paper.

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 see the contents of the shared dictionary as it is being updated, and can even add, delete, or change values to see what effects it will have.

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;
- 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 [22] as a wrappers 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 [21].

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.

EXPERIENCES WITH GT/SD

GT/SD stakes out the middle ground of groupware toolkits. For those building prototypes, its basic networking

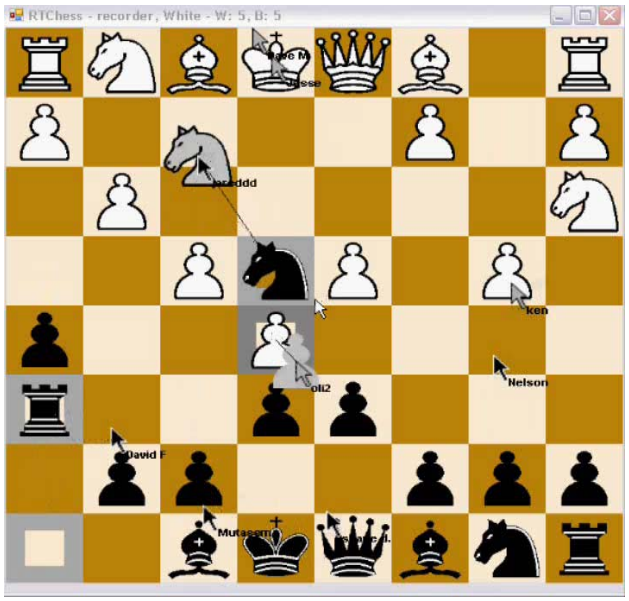

Figure 3: RTChess. Ten players are represented with telepointers and name tags. Players jareddd and oli2 are moving pieces; the black rook has just been moved.

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. GT/SD has been used to develop a number of sophisticated groupware systems within our labs, and by other researchers at different organizations.

Using GT, we have built several group games, a distributed chalkboard application; and an advanced chat client featuring screen-sharing. As a testament to GT's simplicity, a first-year student with no prior networking development expertise used GT to develop two distributed games over a summer term. Similarly, the Shared Dictionary is the basis for many quite different applications. One example includes our Shared Phidgets project, a very sophisticated toolkit that connects hardware, sensors and actuators located anywhere on the internet, and that visualizes data as it moves around the network. It uses the dMVC model to collect, store and propagate values between these hardware devices [14].

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 [9]. 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.

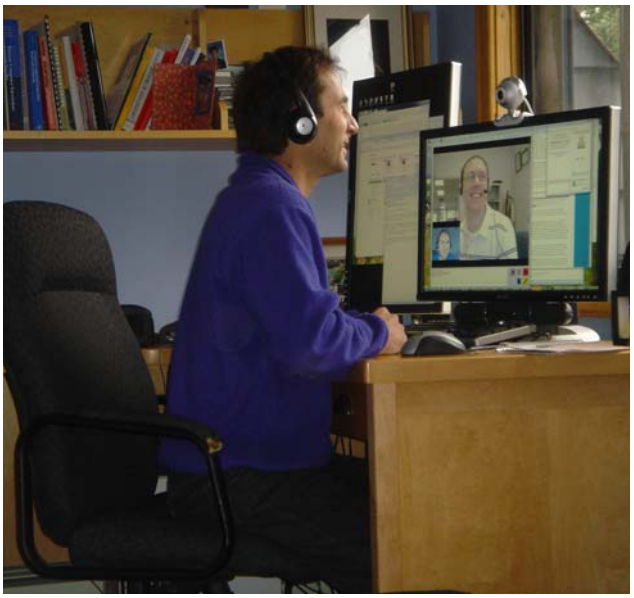

Figure 4: ME-dia space home node. The distant person is visible in the large video, while the local person's sent video is mirrored at the lower left to provide feedback

RTChess was built using GT. It totals 5850 logical lines of code (LLoC), of which the GT-related portions account for 524 LLoC. The application is structured using two channels. The first channel carries presence information, such as telepointers, and is configured with TelepointerLike 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 a server to arbitrate commands: clients make requests of the server, and the server serializes the messages and 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 illustrate GT's extensibility, we implemented a custom marshaller (in 317 LLoC) to ensure minimally-sized packets. 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 [24], 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.

DISCUSSION AND CONCLUSION

GT/SD solves many problems facing groupware developers, but not all of them. We fully realize that many significant barriers are left in groupware development that should be handled at the toolkit level. Examples include: application-level concurrency control [6], deeper QoS support, inclusion of groupware widgets [12,18], tailoring network communications to different architectures, network address translation, firewall traversal, secure connections, network monitoring, voice over IP, video streaming, real-time protocols for audio and video, support for peer to peer networks, and so on.

The above list sets the research agenda for our future GT/SD research and for future toolkits built by other researchers. However, its incompleteness does not take away from the current version of GT/SD. Our experiences with the toolkit show that we have succeed in meeting the design goal stated earlier – that basic groupware should be trivial, good network performance should be only slightly more difficult, and full control should be possible for those that want it. GT/SD delivers on this philosophy by dramatically simplifying network setup and data sharing, simplifying vital aspects of network control, yet still giving full power to application developers when needed.

ACKNOWLEDGMENTS

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

Software Availability. See hci.usask.ca/research/gt/ or 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. Crowley, T., Milazzo, P., Baker, E. & Forsdick, H. MMConf: An infrastructure for building shared multimedia applications. *Proc. ACM CSCW*, (1990)
3. 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)
4. 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)
5. Greenberg, S. Toolkits and interface creativity. *J. Multimedia Tools and Applications*, 32(2), Springer, 139-159, (2007)
6. 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)
7. 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)
8. Graham, T. N., Urnes, T. & Nejabi, R. Efficient distributed implementation of semi-replicated synchronous groupware. *Proc. ACM UIST*, 1-10, (1996)
9. Gutwin, C., Barjawi, M. & de Alwis, B. Chess as a twitch game: RTChess is real-time multiplayer chess. Demonstration at *ACM CSCW*, (2008).
10. 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)
11. Gutwin, C., Benford, S., Dyck, J., Fraser, M., Vaghi, I. & Greenhalgh, C. Revealing delay in cCollaborative environments, *Proc. ACM CHI*, 503-510, (2004)
12. Hill, J. & Gutwin, C. The MAUI toolkit: Groupware widgets for group awareness. *Comp. Supp. Coop. Work*, 13(5-6), 539-571, (2004)
13. Kendall, S.C., Waldo, J., Wollrath, A. & Wyant, G. A note on distributed computing. Tech. Rep. TR-94-29, Sun Microsystems Inc., (1994)
14. Marquardt, N. & Greenberg, S. Distributed physical interfaces with Shared Phidgets. *Proc. ACM Tangible and Embedded Interaction*, 13-20, (2007)
15. Patterson, J.F., Day, M. & Kucan, J. Notification servers for synchronous groupware. *Proc. ACM CSCW*, 122–129, (1996)
16. Pedersen, J.B. & Wagner, A. Sequential debugging of parallel message passing programs. *Proc. Commun. in Comput.* (CIC), 55–61, (2000)
17. Rodden, T., Mariani, J. & Blair, G. Supporting cooperative applications. *Comp. Supp. Coop. Work*, 1(1-2), 41–67, (1992)
18. Roseman, M. & Greenberg, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM TOCHI*, 3(1), 66–106, (1996)
19. 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)
20. Schneiderman, B. Creativity support tools: A grand challenge for HCI researchers. In *Engineering the User Interface* (M. Redondo, Ed.), Springer, (2009)
21. Stuckel, D. & Gutwin, C. The effects of local lag on tightly-coupled interaction in distributed groupware. *Proc. ACM CSCW*, 447–456, (2008)
22. Tanenbaum, A.S. *Computer Networks*. Prentice-Hall, 4th edition, (2003)
23. 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)
24. Voida, A., Voida, S., Greenberg, S. & He, H. Asymmetry in media spaces. *Proc. ACM CSCW, 2008.*
25. 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)

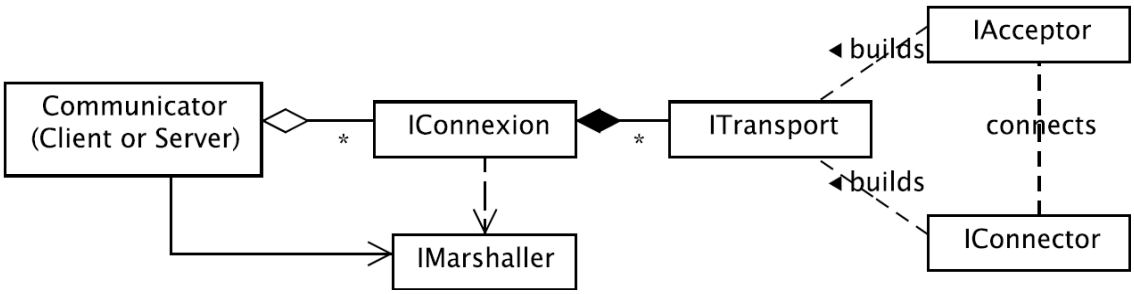

Figure 1: High-level design of the GT framework