.GT: A Groupware Toolkit for C#

Brian de Alwis
bsd@acm.org

ABSTRACT
Building and experimenting with groupware applications is made complicated by having to deal with networking quirks, issues of concurrency, and non-extensible communication mechanisms. We have implemented a toolkit for to simplify the development of groupware applications in C#/.NET that aims to reduce the amount of code necessary to prototype an idea.

Note: This document is for GT 3, and is meant to complement, not replace, the system documentation included with GT.

Keywords
Toolkits, network programming, extensible systems

INTRODUCTION
Implementing a communication toolkit is time consuming. Network programming is rife with managing complex details, ranging from low-level, almost trivial details, such as remembering to increase a TCP listening socket's queue size and disable Nagle's algorithm when sending fixed-size TCP packets, to higher-level implementation details such as multiplexing multiple logical connections across a single network connection, handling sudden disconnects, and dealing with concurrency.

In this paper, we describe a toolkit that aims to simplify building groupware systems on the .NET platform. This toolkit, called GT [1], is 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. Its modular design separates the different concerns involved in network communication, supporting experimenting with different components such as new communication protocols, object marshallers, and quality of service specifications.

GT: THE GROUPWARE TOOLKIT FOR C#
GT is a toolkit 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 can be supported using its connexion model.

GT Design Overview
GT exports a notion of connection-oriented, message-based communication between two endpoints. A logical connexion is divided into a set of 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; these classes provide the bulk of the programmer-facing APIs. These client and server instances send and receive messages with other remote endpoints through *connexions*. A connexion is a logical grouping of the different *transports* that connect to the same logical endpoint. Each transport represents an established network communication protocols. A connexion selects a transport for sending a message that best meets the message's delivery requirements. A *marshaller* transforms a message to and from a byte array. The *connexion* uses a packet scheduler to order how messages are marshalled and assembled into a packet. The *acceptor/connector* design pattern is used to separate establishing a transport from the actual delivery of packets [3].

Programmers can 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.

GT supports communication over multiple transports with differing transport characteristics. GT provides three transports: a TCP based transport, providing reliable and

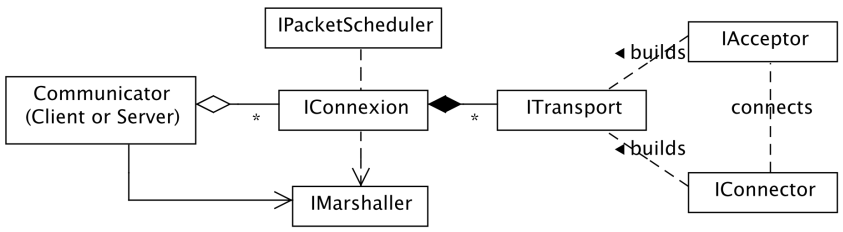

Figure 1: High-level design of the GT framework

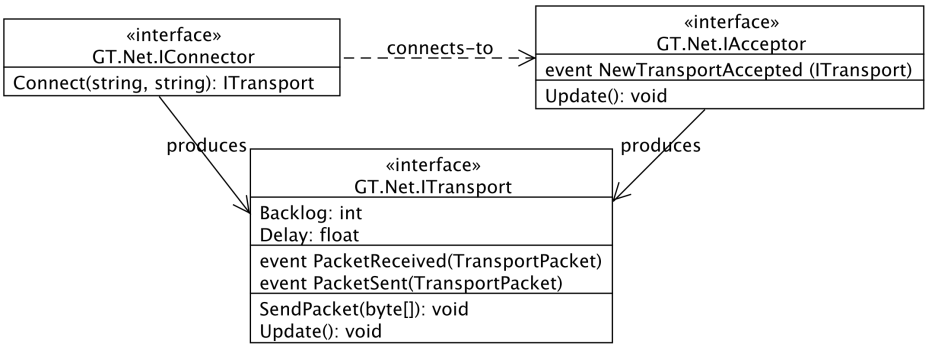

Figure 2: Lowest level of the GT stack, pertaining to the establishment of connections and sending bytes between network-level endpoints

ordered delivery; a UDP-based transport, providing unreliable and unordered delivery; and a sequenced UDP transport, providing unreliable but sequenced delivery. There is also a local transport, providing reliable and ordered intra-process delivery using a shared queue, which is useful for testing purposes.

Sending Messages with GT

GT uses a non-blocking polling design (except for establishing connections), suitable for use in a single-threaded application. Although GT is thread-safe, many developers choose to structure their applications using single-threaded operation to help avoid possible complications arising from concurrency issues.

GT hides the details of the underlying communication mechanisms, such as TCP or UDP, from the developer. Messages and channels are instead annotated with their delivery requirements that are used to find the transport whose characteristics best meet the delivery requirements. These delivery requirements include reliability, message ordering, and the message timeliness. Specifying requirements frees a programmer from having to remember specific quality of service (QoS) characteristics of the networking mechanisms, and avoids embedding implicit knowledge in the source code.

The receipt of messages is handled through GT's fine-grained events, implemented using the C# event mechanisms. These events provide notification of messages being received and sent, as well as progress as messages work their way through the GT. These events can be hooked into to provide diagnostic functions; we discuss this further later in the paper. GT also maintains details of round-trip communication times for the various underlying communication mechanisms, and the message backlogs.

Design

GT is designed as a three-level protocol stack, separating the different concerns involved in transporting messages across a network. Its modular design supports experimenting by replacing or wrapping well-defined components. GT's design has also sought to separate policy decisions from the mechanisms required to implement the different policies.

Network-Level Communication Abstraction

At the bottom of the stack are abstractions for sending and receiving byte arrays, referred to as packets, across a particular communication mechanism between two communication endpoints (e.g., a server and a client across a TCP socket). Each communication mechanism is represented as instances of ITransport, IAcceptor, and IConnector (see Figure 2). An ITransport object represents an established connection between two communication endpoints. GT ships with four implementations of ITransport: a TCP based transport, providing reliable and ordered delivery; a UDP-based transport, providing unreliable and unordered delivery; a sequenced UDP transport, providing unreliable but sequenced delivery; and a local transport, providing reliable and ordered intra-process delivery using a shared queue. The negotiation of a connection between two communication endpoints ITransports are established using the acceptor/connector design pattern [3]. Each of the transports described above have corresponding IAcceptor and IConnector instances; acceptors listen and process incoming connection requests, and connectors issue connection requests to a particular acceptor.

Messaging Between Clients and Servers

The middle level of the stack (Figure 4) describes a logical grouping of the different communication mechanisms between a client and a server. This logical connection is represented by an IConnexion. The IConnexion provides for sending and receiving typed messages between to the remote endpoint, and selects an appropriate ITransport based on delivery requirements provided with each message. IConnexions use a marshaller for converting a message, represented as a class of Message, to a byte form.

Connexion Management

The top of the stack (Figure 3 and Figure 5) provides the expected programmer-facing interfaces to GT. Client and Server provide different types of interfaces for sending and receiving messages. Client instances are programmed

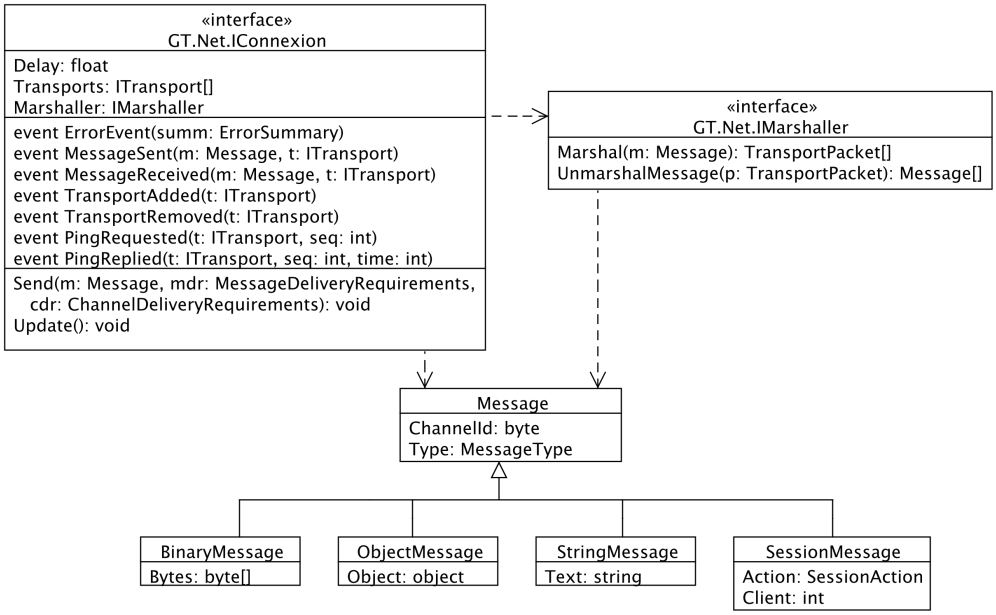

Figure 4: Intermediate level of the GT stack, representing the grouping of transports as a logical connexion for sending messages between a client and server

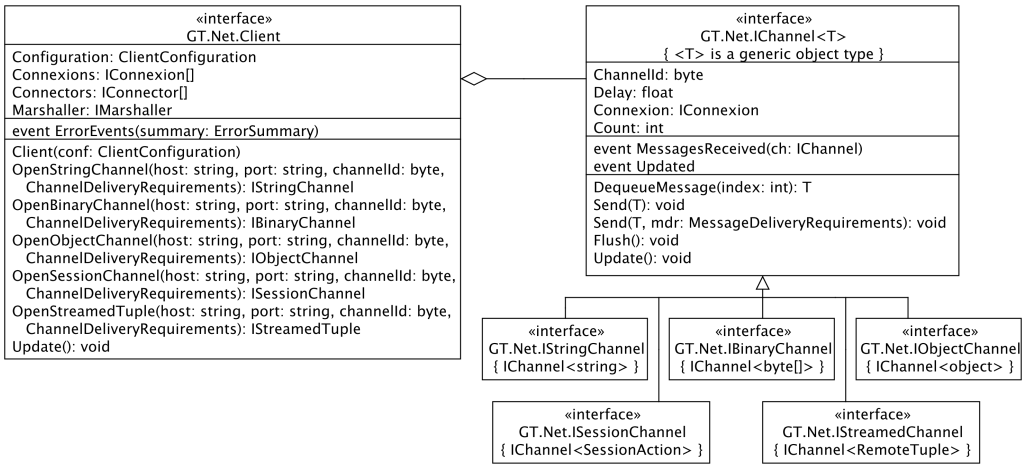

Figure 3: Client-side of the upper level GT stack: Connexion management.

by requesting a typed channel for a particular host/port pair and a particular channel id. A connexion is looked up for the host/port pair, and created if not found using the configured connectors, and the channel instance is used for sending and receiving objects of the appropriate type (e.g., byte arrays for a binary channel, strings for a string channel). We show examples below. The Server instance is a more event-driven mechanism, featuring C#-style events for being notified of the receipt of a new message, and new and removed connexions.

GT's design has sought to separate policy decisions from the mechanisms required to implement the different policies. This has been accomplished primarily by isolating policy choices or classes that define policy to a central Configuration object. Both Client and Server are configured through their respective ClientConfiguration and ServerConfiguration objects. These configuration objects define the particular acceptors and connectors to use, the marshaller instance, and the default channel delivery requirements objects. Programmers can subclass the configuration objects to override the choices made.

Configuration

GT separates the mechanisms for communication from the policies through use of a user-supplied a Configuration object. There is a configuration object for each of a client and a server, respectively defined by the abstract ClientConfiguration and ServerConfiguration objects. The configuration objects are responsible for

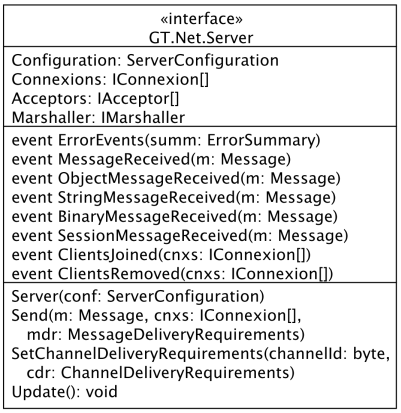

Figure 5: Server-side of the upper-level GT stack.

provide configured implementations of required objects. For example, the server configuration is responsible for instantiating and configuring the acceptors to be listened on by a server, such as the port listened on by IP-based acceptors. Similarly, the client configuration is responsible for configuring the connectors used by a client.

Ready-to-run configurations are provided as the DefaultClientConfiguration and DefaultServerConfiguration classes. These classes have reasonable defaults that will fit the needs of most prototyping efforts. As these definitions may change between GT releases without warning, we encourage that a customized configuration is created when deploying an application.

EXAMPLES

Having provided a high-level overview of the GT, we demonstrate the ease of using GT to build a functional system through a set of examples. The first example defines a simple server implements a frequently used pattern for groupware systems, called a client-repeater. The remaining examples assume the use of this ClientRepeater.

Example 1: A ClientRepeater Server

We have found in our use of GT that a successful pattern for building groupware applications features a simple server that simply repeats all incoming messages to all connected clients. We have reified this pattern as a standard server application shipped with GT, called ClientRepeater. The following presents a simplified but fully-functional variant; the shipping version has some minor optimizations and advanced features, such as logging, that are beyond the scope of this paper.

The ClientRepeater requires a Server instance, which is configured by:

```
void Main(string[] args) {
  Server server = new Server(
    new DefaultServerConfiguration(9999));
  server.ErrorEvent +=
    summ => Console.WriteLine(summ);
  server.MessageReceived += s_MessageReceived;
  server.ClientsJoined += s_ClientsJoined;
  server.ClientsRemoved += s_ClientsRemoved;
  server.StartListening();
}
```

The DefaultServerConfiguration configures a TCP and a UDP acceptor to listen on port 9999. This snippet uses C#'s event mechanisms to request notification of four different events. The Server.ErrorEvent event is called upon the occurrence of some event of possible significance to the developer; the summary object provides an assessment of the severity and details about the event. The MessageReceived, ClientsJoined, and ClientsRemoved, are triggered in response to incoming messages and connection changes, and are described shortly. The final line calling Server.StartListening() invokes an infinite loop to process incoming events. GT generally separates the creation and configuration of an object from starting the object.

The Server's MessageReceived event is triggered whenever a new message has been received. The ClientRepeater broadcasts each received message to all connected clients:

```
void s_MessageReceived(Message msg, ...) {
  server.Send(msg, null, ...);
}
```

The use of null for the connexions means to send to all connexions. The ClientsJoined and ClientsRemoved events are also broadcast to all clients:

```
int SessionUpdatesChannel = 0;

void s_ClientsJoined(IConnexion[] list) {
  foreach(IConnexion newConn in list) {
    server.Send(
      new SessionMessage(SessionUpdatesChannel,
          newConn.Identity, SessionAction.Joined),
      null, ...);    } }

void s_ClientsRemoved(IConnexion[] list) {
  foreach(IConnexion oldConn in list) {
    server.Send(
      new SessionMessage(SessionUpdatesChannel,
        newConn.Identity, SessionAction.Left),
    null, ...); } }
```

Session updates are sent on channel 0 by default. All other messages are broadcast on the same channel on which they were received.

Just this code implements a simple client-repeater that (i) broadcasts all incoming messages to all connected clients, regardless of the message type (e.g., binary, string, object) and the channel upon which they were received, and (ii) notifies connected clients of session updates on channel 0. More sophisticated servers can be implemented in a similar fashion.

Example 2: Chat Client

Our first client example demonstrates the ease of creating a simple chat client, as shown in Figure 6. This client assumes the use of a client-repeater server such as the one described in the previous example. When the user has finished composing a new chat message, the chat client

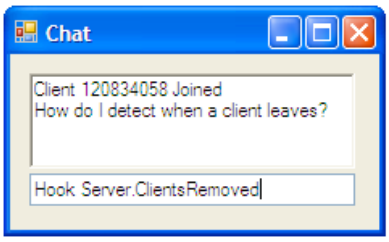

Figure 6: The Client Chat window. The upper text box is the transcriptBox, and the lower text box is the composedBox.

sends the newly composed message (a text box named composedBox) to the client-repeater. The client appends the text from incoming messages to its conversation transcript (a text box named transcriptBox). Due to space concerns, we are unable to show the actual UI code.

This client requires a Client instance and two channels:

```
Client client;
IStringChannel chats;
ISessionChannel updates;
```

We also define two constants to designate the channels carrying session update events (with the default channel as from the ClientRepeater) and the actual chat messages:

```
const int SessionUpdatesChannelId = 0;
const int ChatMessagesChannelId = 1;
```

Our chat client must first create and start the GT Client instance; this is often performed from the UI Form constructor:

```
client = new Client(new DefaultClientConfiguration());
client.ErrorEvent += delegate(ErrorSummary summ) {
  Console.WriteLine(summ); }
client.Start();
```

The Client.ErrorEvent event serves the same purpose as the Server.ErrorEvent seen in the ClientRepeater example. The call to Client.Start() places the instance into a running state.

Having started the client instance, we open channels to send and receive chat messages and to receive session updates. We assume that the UI has somehow prompted the user to obtain a host/port endpoint for the client-repeater. Given these endpoint details, we allocate two channels:

```
chats = client.OpenStringChannel(host, port,
  ChatMessagesChannelId,
  ChannelDeliveryRequirements.ChatLike);

updates = client.OpenSessionChannel(host, port,
  SessionUpdatesChannelId,
  ChannelDeliveryRequirements.SessionLike);
```

These channels specify predefined delivery requirements for different types of messages, such as the ordering requirements, the reliability requirements, and the aggregatibility of the channel [2]. Messages sent with ChatLike must be delivered in the order sent, whereas those sent with SessionLike may be delivered out of order. The delivery requirements are used to select the minimal transport capable of meeting these requirements when sending a message on a particular channel.

In this example we will poll the channels for messages from a UI-compatible periodic timer. This approach avoids the problems from UI cross-threading issues by having all network communication be performed on the UI thread. When messages are available, we update the transcript:

```
void timer_Tick(object sender, EventArgs e) {
  client.Update();          // process all connexions
  string chat;
  while((chat = chats.DequeueMessage(0)) != null) {
    transcriptBox.Text += chat + "\n";
  }
  SessionMessage sm;
  while((sm = updates.DequeueMessage(0)) != null) {
    transcriptBox.Text += "Client " + sm.Client + " "
      + sm.Action + "\n";
  }
}
```

We hook the composedBox's key-pressed event to determine when to send the message being composed:

```
void composedBox_KeyPressed(object sender,
    KeyEventArgs e) {
  if(e.KeyCode != Keys.Enter) { return; }
  messages.Send(composedBox.Text);
  composedBox.Text = "";
  e.Handled = true;
}
```

The outgoing message will be sent to the client-repeater, which will broadcast the message to all clients, including this client. Thus the chat client does not add its own message directly to its conversation transcript, but waits until its message has been received from the client-repeater's broadcast.

Example 3: Telepointers

Our final GT example demonstrates for sharing telepointers between clients, as shown in Figure 7. Due to space concerns, we are unable to show the UI code. In this example, each client uses a *streamed tuple* to update their mouse position. A streamed tuple accumulates changes and periodically sends the latest change to the client-repeater to be broadcasted to all connected clients. These other clients are notified of an update to another client's tuple through an event providing the client identity and the new tuple contents.

Similar to the chat client, this client also requires a Client instance and two channels. The first channel is used for session updates, and the second channel is used for sending updates of client telepointer positions:

```
Client client;
IStreamedTuple<int,int> coords;
ISessionChannel updates;
IDictionary<int, Point> telepointers;
```

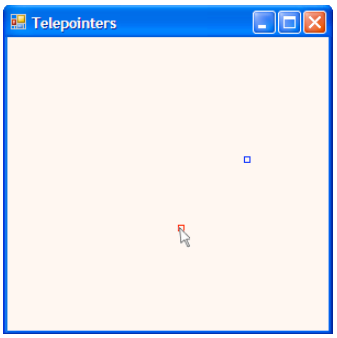

Figure 7: A simple telepointer example where the current desktop's cursor position is shown in red, and the other desktop's cursor position is shown in blue.

We also define two constants to designate the channels carrying session update events, corresponding to the client-repeater, and the actual telepointer updates:

```
const int SessionUpdatesChannelId = 0;
const int TelepointersChannelId = 1;
```

Similar to the previous example, our telepointer client must first create and start the GT Client instance, such as from its constructor:

```
client = new Client(new DefaultClientConfiguration());
client.ErrorEvent += delegate(ErrorSummary summ) {
  Console.WriteLine(summ); }
client.Start();
```

Having started the client instance, we open channels to send and receive telepointer updates and to receive session updates. Again, we assume that the UI has somehow prompted the user to obtain a host/port endpoint for the client-repeater. We then allocate two channels:

```
updates = client.OpenSessionChannel(host, port,
  SessionUpdatesChannel,
  ChannelDeliveryRequirements.SessionLike);

coords = client.OpenStreamedTuple<int, int>(host, port,
  TelepointersChannel,
  TimeSpan.FromMilliseconds(50),
  ChannelDeliveryRequirements.AwarenessLike);
```

The above causes the streamed tuple coords to update the server every 50 ms with its latest values. Any changes between these notifications will be noted but not propagated.

In this example, we use an event-driven mechanism to be notified of incoming messages, as compared to the polling approach used in the previous example. These events are automatically triggered as a result of calling Client.Update():

```
updates.MessagesReceived +=
  updates_SessionMessagesReceived;

coords.StreamedTupleReceived +=
  coords_StreamedTupleReceived;
```

Notification that a client has left the session causes the client's telepointer to be removed:

```
void updates_SessionMessagesReceived (
    ISessionChannel channel) {
  SessionMessage m;
  while ((m = channel.DequeueMessage(0)) != null) {
    if (m.Action == SessionAction.Left) {
      telepointers.Remove(m.ClientId);
    }
  }
}
```

Notification of a streamed tuple update causes the corresponding client's telepointer to be updated:

```
private void coords_StreamedTupleReceived(
    RemoteTuple<int, int> tuple, int clientId) {
  telepointers[clientId] = new Point(tuple.X, tuple.Y));
}
```

Whenever the local client's mouse is moved, we update our streamed tuple by directly assigning the tuple's fields; the tuple is periodically flushed, sending the latest to all clients:

```
private void Form1_MouseMoved(object sender,
    MouseEventArgs e) {
  coords.X = e.X;
  coords.Y = e.Y;
}
```

Finally, this example also uses a UI-compatible periodic timer to avoid potential UI cross-threading issues:

```
void timer_Tick(object sender, EventArgs e) {
  client.Update();          // process all connexions
  UpdateTelepointers();
}
```

UpdateTelepointers() is responsible for drawing the telepointers at the appropriate locations using the values in telepointers.

EXTENDING AND CUSTOMIZING GT

Hooking GT Events

GT features a number of events that can be used to instrument what is occurring. GT provides events to notify of:

- connexions added and removed;
- transports added and removed;
- an Update() cycle has completed;
- a ping has been requested and a response received;
- a message has been sent or received;
- a packet has been sent or received;
- some error has occurred.

For example, the PingBasedDisconnector hooks the PingRequested and PingResponsed events for all IConnexions to automatically disconnect transports whose remote side does not respond within some configured time. This disconnector also uses GT events to automatically install itself on any new connexions. The disconnector is installed using code like:

```
Client c = …;
PingBasedDisconnector pbd =
```

```
PingBasedDisconnector.Install(c,
  TimeSpan.FromSeconds(30));
```

Marshalling Messages

GT's architecture allows new marshalling schemes to be created by replacing a pluggable marshaller implementing the GT.Net.IMarshaller interface. Each marshaller must provide a description of its marshaled format through the string property IMarshaller.Descriptor. This is expected to be a colon-separated set of strings that progressively describe the container format of messages that are marshalled using the marshaller. This descriptor is provided to the remote marshaller to determine how to unmarshal messages.

The GT-provided marshallers all use a common primitive container format called "Lightweight Message Container Format v1.1" (or LWMCFv1.1) that takes 6 bytes. The first byte describes the message type (GT.Net.MessageType). The second byte describes the channel for that message. The next 4 bytes form a uint32 describing the message size in a little-endian format. The remaining bytes contain the marshalled message bytes.

Transports and TransportPackets

GT uses transports implementing ITransport to send and receive packets, a sequence of bytes, between two endpoints. Transports implement a very simple operation set, namely send a packet and receive a packet.

These bytes are provided as an instance of TransportPacket. TransportPackets provide an efficient way to manage byte sequences, including splitting and joining different byte sequences. They use a managed-memory scheme to avoid creating garbage, and implement a customized reference-counting implementation to share byte sequences between multiple users. It is considered to be a criticial error if a TransportPacket is accessed after having been disposed.

Transports are expected to dispose of any TransportPacket instances provided to ITransport.SendPacket(). Having received a packet, a transport triggers its PacketReceived event and then dispose of the packet; if any listeners of the event wish to use the packet beyond the lifetime of the event, then the listener is responsible for retaining the packet.

On an fatal error in SendPacket(), a transport may throw a TransportError exception. Such an exception will cause the transport to be disposed. Recoverable errors or warnings should be reported using the ErrorEvent mechanism.

DEBUGGING & TESTING SUPPORT

GT has a number of features to support debugging and testing an application. The GT.Utils namespace has several smaller classes for formatting data, such as ByteUtils.HexDump().

GT Events

GT features a number of events that can be used to instrument what is occurring. We have had some success understanding a distributed application's behavior by playing sounds for particular events such as when transports are added and removed, when a connexion is added or removed, when a client or server has been updated, and on ping request and response events (to confirm traffic is reaching the opposite side).

Logging

GT 3.0 now uses the Common.Logging framework for logging messages. GT's information is mapped in a straightforward manner to Common.Logging's Trace, Info, Warn, and Error levels.

Common.Logging provides logging adapters for more full-featured logging packages such as log4net, NLog, and Microsoft's Enterprise Library logging facilities. It is also very easy to write a custom adapter to use another some other logging mechanism.

Common.Logging allows selecting a logging implementation at runtime, either programmatically or via the .NET App.Config mechanism. By default, Common.Logging uses the NoOpLoggerFactoryAdapter which suppresses all output. The ClientRepeater demonstrates using the App.Config mechanism to select the ConsoleOutLogger to log event to the console.

A final, small note on deploying and configuring applications using GT and Common.Logging. If you plan to install the GT merge-module into the GAC (which contains the GT DLLs as well as the Common.Logging DLL), and your application explicitly configures Common.Logging through the App.Config mechanism, then you must explicitly specify the full version, culture, and public key tokens in your App.Config file. Otherwise your application is likely to fail due to a somewhat confusing assembly binding error. See the ClientRepeater's App.Config for an example of specifying the full version information.

Millipede-Style Debugging

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, in researching debugging parallel program in their in Millipede system [1], had a key insight 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, GTMILLIPEDE. On startup, GT client and server processes check this variable for one of three values: "record:*file*" to record the incoming and outgoing messages for this session to a file, "replay:*file*" to replay the messages from a file, and left unset or to "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.

When programs are non-deterministic for reasons other than GT communication (e.g., keyboard and mouse input or timeouts), our debugging support may not exactly reproduce the recorded behaviour. There are, however, workarounds for these cases, such as integrating the sources of the non-determinism into the GT-Millipede framework. Each source must have a unique descriptor, allocated by registering the source with MillipedeRecorder.GenerateDescriptor(). Depending on whether the recorder is recording or replaying, the source either records the current value of the source, or checks for a value as previous recorded.

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 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.

Network Emulation

GT 3.0 includes a NetworkEmulationTransport to assess how an application behaved under different simulated network conditions. This transport is a transport wrapper, typically wrapped around another transport through the Configuration.ConfigureNewTransport(). The transport features several properties to set the supported conditions, including transmission delay, packet loss, and packet reordering. GT also includes implementations of a leaky-bucket and token-bucket traffic-shaping algorithms [5] as transport wrappers.

CONCLUSIONS

This article has presented an overview of GT and SD, highlighting the simplicity of the programming model. GT provides well-separated components for building a message-oriented framework.

For further tips, tricks, and FAQs on developing and debugging with GT and .NET, please see the GT website at http://hci.usask.ca/research/gt/.

REFERENCES

[1] B de Alwis, C Gutwin, S Greenberg (2009). GT/SD: Power and Simplicity in a Groupware Toolkit. To appear in Proc. of ACM SIGCHI Symposium on Engineering Interactive Computer Systems (EICS).

[2] J Dyck, C Gutwin, TCN Graham, D Pinelle (2007). Beyond the LAN: Techniques from network games for improving groupware performance. *In* Proceedings of the International Conference on Supporting Group Work (GROUP), pp. 291–300.

[3] JB Pedersen, A Wagner (2000). Sequential Debugging of Parallel Message Passing Programs. *In* Proceedings of the International Conference on Communication in Computing (CIC), pp. 55-61.

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

[5] AS Tanenbaum (2003). *Computer Networks*. Prentice-Hall, 4th ed.

APPENDIX: DELIVERY REQUIREMENTS AND SENDING REQUIREMENTS

This section is a short note addressing the difference between delivery requirements and sending requirements.

GT supports specifying a set of delivery requirements (DR) for each channel. It is important to note that these DRs are used *only to select a transport* for messages sent on that channel *to a particular destination*. The DRs do not describe the delivery semantics for a message in relation to other messages sent within the network (i.e., providing an absolute or causal ordering of messages).

That is, a channel with Reliable and Ordered requirements means that the messages sent on this channel should be sent on a transport that provides reliable and ordered delivery. It does *not* mean that the messages sent on this channel must be delivered in order on the other side; otherwise it's not clear what semantics should be attached to a message with Ordering = Sequenced but Aggregatability = Immediate that is sent after a set of messages with Ordering = Ordered. But messages sent on the same channel with Ordering = Sequenced to the

same destination should select the same transport and be received in the same order sent.

The DR for a channel may be overridden on a per-message basis. Providing an override is equivalent to temporarily changing the channel's DRs, sending the message on the channel, and then reverting the channel's DRs to the original values.

Packet schedulers are responsible for aggregating marshalled messages to be sent across the best transport meeting each message's DR. There is a tension between finding the best transport that meets the requirements of a message and keeping the transport channels as full as possible. This tension is resolved by each individual packet scheduler.