Design and Implementation of a Reliable Group Communication Toolkit for Java

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

Abstract

With the advent of Java, object-oriented technology has become mainstream. To make object-oriented applications reliable requires supporting technologies to be available in object-oriented form too, avoiding an impedance mismatch between different technologies as frequently seen in the database world.

Existing approaches to provide reliability for object-oriented applications have assumed that reliability can be added to an application after the fact, and that it should be as transparent as possible, resulting in frameworks that impose a pre-defined reliability strategy upon users. However, many applications do not need or do not want full transparency and therefore cannot use these frameworks.

A more promising approach to achieve reliability seems to be the *toolkit approach*. Toolkits provide a number of fine-grained classes of different abstraction levels, matching the varying requirements of applications. Such an approach does not force any specific kind of thinking on the user, but leaves it to the programmer which classes to choose and which not.

Process groups [Bir96] are seen as a viable solution to tackle reliability issues in distributed systems. The core idea is to have critical components created redundantly so that when a component fails other components can take over (replication). Additionally, groups can be used to load-balance service requests over a number of servers, and to divide larger problems into smaller ones which can be processed concurrently by different process group members. A major task of group communication is to support quality of service properties on the communication between group members, e.g. ensuring that messages sent to the group are received by all members in the same, well-defined order.

A Java-based toolkit for reliable group communication is currently developed at Cornell University. This paper presents its design and implementation.

1 Motivation

In today's distributed systems, the concept of groups is very common: Usenet news uses discussion groups to categorize areas of common interests, email sent to a distribution list will be sent to all members subscribed to that list, objects in a management domain behave according to a certain policy set forth for that domain, the MBone sends live data streams to all members subscribed to the same channel and most modern air planes have their mission critical systems replicated to tolerate failures.
In the latter example, a replicated service provides availability or fault-tolerance to its clients as long as there is at least one member (or a majority of members) left in the group. At the same time, upgrading such a service is easy: each member is shut down, upgraded and restarted in turn without negative impact on the availability of the service as a whole.
Furthermore, groups are also used to load-balance requests between members of a group, reducing the load on a single process, and to break a large task into smaller pieces, distributing them to the group members which can work on it concurrently and - when done - assembling the results and returning them to the client.
With the advent of Java, the need has arisen to make group communication toolkits available which a) integrate seamlessly with the language and b) exploit Java's advantages: portability allows a Java program to run unchanged on any architecture, dynamic code loading gives the client the possibility to download both classes and instances at runtime (e.g. a whole protocol stack), and reflection makes certain kinds of designs feasible, as will be explained later in the paper.
Previous work in reliable group communication has produced a number of toolkits ([MPS92, BR94, VRBM96, GG97a, Hay98, MS97]). However, most of these cannot be accessed from Java, and – if they do – compromise Java's platform independence by requiring some platform-specific executable or process. Also, due to the fact that most of them are not written entirely in Java (with the exception of iBus), their code cannot be dynamically downloaded to a client, as is the case when downloading an applet. The iBus toolkit [ibu98] is one of the few group communication toolkits available (as of November 98) that are written entirely in Java. Whereas iBus' primary goal is reliable message dissemination based on the publish-subscribe philosophy and focuses less on abstractions, our goal is to find and catalog *patterns* in group communication settings. iBus has been integrated with our work (see below), so it can be used as a reliable group transport.[1] On the other hand, our collection of patterns could be easily ported to work directly over iBus.
This paper presents a reliable group communication toolkit, called JavaGroups, written entirely in Java. It is modeled after Ensemble [Hay98] and Horus [VRBM96], but tries to exploit the benefits of Java. The goals we pursue are manifold: first, having a pure Java implementation allows us to experiment with new designs for group communication. Second, such a toolkit serves as a test-bed for the development of new Java-based protocols. Finally, a major focus of this work is to find recurring design elements (patterns), categorize them and capture them in class form. We are mainly interested in two kinds of patterns: structural and behavioral. Structural patterns are about how objects and classes are composed to form larger structures and capture group communication elements such as protocol layers or messages. Behavioral patterns deal with recurring communication exchanges, such as the transfer of state from an existing member to a joining one, or a protocol to elect a new leader when the old one has died.
The structure of this paper is as follows: first, an overview of the architecture is given. Then we present the JavaGroups interface as seen by a programmer who wants to build group communication applications. A number of (mostly structural) patterns is described in this section. Finally, a brief overview of the protocol stack architecture is given.

[1] A secondary goal of our work is to provide a reliable group communication substrate, cf. section 4.

2 Architecture

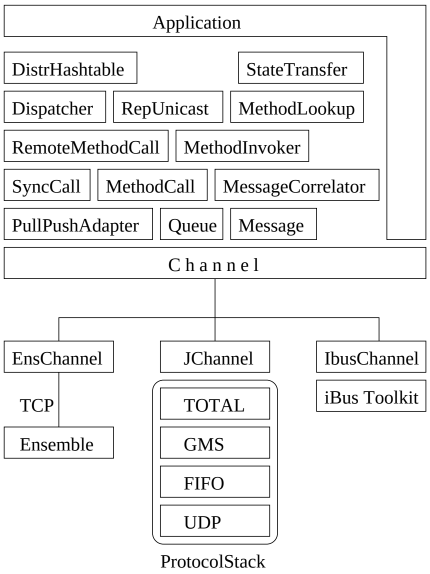

Figure 1: Overview of the JavaGroups Architecture

The architecture of JavaGroups can be divided into two parts: the API, i.e. the interfaces used by an application to *make use* of group communication, and the classes used to *implement* group communication, i.e. the protocol stack.[2] The API will be discussed in section 3, the protocol stack architecture in section 4.

Central to the JavaGroups architecture is the concept of a *channel*, which can be seen as a handle to a group. Each channel has a name and channels with the same name form a process group. The channel interface is very small and offers only the basic functionality to connect to a group, send and receive messages, get the members of a group and to be notified when members join or leave. Applications may use this interface if they require efficient low-level socket-like group communication functionality.

However, programming at this low abstraction level requires application code to handle more sophisticated tasks, such as correlating responses with their corresponding request or simulating synchronous message calls on top of the inherently asynchronous channel.

One of the goals of JavaGroups is to provide *patterns* [GHJV95] to handle tasks on a higher abstraction level than the channel. They offer a collection of *structures* and *algorithms* often encountered in group communication in the form of Java classes. Structural patterns may for example be a synchronous message call as represented by `SyncCall` in fig. 1. Algorithmic (or behavioral) patterns capture frequently occurring communication schemes between participants in group communication settings, such as for example the election of a new coordinator when the old one has crashed, or the implementation of a state transfer protocol to bring a joining member up to date. They may be used both for application and protocol development. Developers are free to pick the level of abstraction best suited for their application.

The value of patterns is that they are thoroughly tested, frequently used pieces of design, captured in a single place (a class). They should be adaptable to suit various needs, and at the same time small and sufficiently independent from other patterns to give the application programmer the freedom to choose which patterns to pick and which not. Small granularity

[2]The protocol stack architecture allows to write new protocols, or change existing ones, and use them without modifying its code.

of patterns gives the user a choice of replacing single patterns if they do not suit the application's need (and cannot be configured to do so) with their own implementation, which is considerably easier than to rewrite large-grain patterns. Finally, use of patterns reduces development time of new applications or protocols.

Some of the patterns offered by JavaGroups require a channel underneath them, whereas others are completely independent from channels. Also, the channel interface is sufficiently simple and small to be ported over different reliable group communication protocol stacks, so that the patterns – which rely only on an implementation of the (abstract) channel interface – can probably be used on most instances of group communication transports.

An actual instance of a channel has to implement the channel interface, providing reliable group communication according to the *properties* required by that channel. Properties are specified in string form when instantiating a channel and define the quality of service a channel has to provide; for example a property may require loss-less, totally ordered delivery of messages, whereas another one may only require unreliable UDP communication between channels. Properties translate to a specific setup of the underlying protocol stack (see section 4).

As shown in fig. 1, JavaGroups currently supports three different kinds of channels. The first (`EnsChannel`) interfaces to Ensemble to provide reliable group communication. The properties defined when creating an Ensemble channel directly map to regular Ensemble properties [Hay98]. JavaGroups spawns an Ensemble process and connects to it via TCP. Client calls to an Ensemble channel are forwarded to that process, making `EnsChannel` essentially a client stub front-end to Ensemble. The advantage of such a solution is that a proven and well-tested reliable group communication toolkit can be used directly from JavaGroups. The downside is that the requirement to install a platform-specific executable may defeat some of the purposes of Java: using this scheme it is for example not possible to dynamically download reliable group communication classes to a client. However, for applications that do not require such functionality, using `EnsChannel` is a feasible way to write reliable group applications.

The second implementation of a channel is based on iBus [ibu98], a pure Java-based toolkit for reliable group communication. Class `IbusChannel` is only a thin veneer over iBus, mapping a JavaGroups channel to an iBus `Stack`.

Finally, our own implementation of a channel (`JChannel`) provides a protocol stack configured according to the properties argument when creating the channel, and consisting of a list of bidirectionally linked protocol layers. It will be shown that some of the JavaGroups patterns can be used profitably to implement protocol layers. The design of `JChannel` is described in more detail in section 4.

3 Patterns

In this section some of the major patterns provided by JavaGroups will be presented. As JavaGroups is work in progress, this is by no means an exhaustive enumeration of all available patterns and we are convinced that during the development of additional protocols and JavaGroups functionality, new patterns will emerge and existing ones may be removed or merged with others to create patterns of higher genericity, moving towards a core collection of stable and frequently used patterns.[3]

[3] A pattern is defined in [GHJV95] as a piece of design used and refined over and over again. Our patterns do not entirely fit this definition as they have not yet been refined by frequent use. We nevertheless use the term as our intent is to eventually live up to the definition. We see the process of finding and refining good patterns as a gradual, open-ended one, similar to finding a good design, which may not happen at the first try.

3.1 Channel

A channel represents the group endpoint over which messages can be sent to all (or a subset) of the group members and over which messages multicast to group members can be received. It is on purpose designed to be as simple as possible, similar to BSD sockets.

Each channel has a name. Channels with the same name form a group, that is, messages sent by a channel will be received by all other channels with the same name.

To use a channel, a client first has to connect to it. When the channel is no longer needed, the user should disconnect from it. Only one client is permitted to be connected to the same channel at the same time.[4] When connected, messages received by the channel will be stored in a queue until `Receive` is called, which returns the oldest message by removing it from the queue. All messages will be deleted from the queue when the client disconnects.

The main reason for the need to connect to a channel before using it is that this serialization of access to the channel 'resource' prevents clients from removing messages from the channel without other clients seeing them. That is, multiple clients of a channel would not see the same sequence of messages.[5]

A client may send a message to (1) a single channel, (2) a number of channels, or (3) all of the channels of the same group using the `Send` (1,2) or `Cast` (3) methods. To find out the other channels in the group, method `GetMembers` can be used, which returns the addresses of all members in the group. Since these addresses will typically vary in their form and content, depending on the transport (i.e. the channel implementation) used, they are only returned in the most general form of `Object`. Each channel implementation has to narrow such an address to the form used by it. Users of JavaGroups must not be concerned about the contents and real class of an address, as this is opaque. They receive addresses as result of method calls and subsequently use them as target addresses in sending messages to a single channel or a set of channels.

Channels use the Half-Sync/Half-Async pattern [SC97] in that they present a synchronous interface to the caller (`Receive`), but internally use asynchronous message reception and message queues to block and awake callers. There are two reasons for using a pull-style for receiving messages on a channel: first, it is similar to what programmers are used to when receiving data from a socket, and second, by not calling user code, channels cannot get blocked by code that takes a long time to complete or that even recursively calls a method of the channel.

If a push-style of message reception is desired, other patterns on top of channels can be used (such as `PullPushAdapter` (3.2) or `Dispatcher` (3.7)).

3.1.1 Example

The sample code in fig. 2 shows how pull-style channels work. The sample creates an Ensemble channel (`EnsChannel`), connects to it and casts a number of messages before exiting. When started, a separate thread is created which retrieves and displays incoming messages from the channel (pull-style). A receiver thread is required when messages need to be sent and received simultaneously.

3.2 PullPushAdapter

This class is a converter (or adapter, as used in [GHJV95]) between the pull-style of actively having to receive messages and the push-style where clients register a callback method which is invoked whenever a message is received.

[4] However, a client may have multiple channels (even with the same name).

[5] This would be different in a push-style channel, where all clients are notified when messages arrive.

```
public class ChannelTest implements Runnable {
    private Channel channel=null;
    private String  props="Gmp:Sync:Heal:Frag:Suspect:Flow:Total";

    public void Start() throws Exception {
        channel=new EnsChannel("TestChannel", props);
        channel.Connect(0);
        new Thread(this).start();
        for(int i=0; i < 10; i++) {
            channel.Cast(new String("This is msg #" + i).getBytes());
            Thread.currentThread().sleep(1000);
        }
        channel.Disconnect();
        channel.Destroy();
    }

    public void run() {
        while(true) {
            try {
                Message msg=channel.Receive(0); // no timeout
                System.out.println(new String(msg.GetBuffer()));
            }
            catch(NotConnected conn) {break;}
            catch(Exception e) {System.err.println(e);}
        }
    }

    public static void main(String args[]) {
        try {new ChannelTest().Start();}
        catch(Exception e) {System.err.println(e);}
    }
}
```

Figure 2: `Channel` sample

A `PullPushAdapter` is always created with a reference to a class that implements interface `Transportable`[6] (e.g. a channel). Clients interested in being called when a message is received can register with the `PullPushAdapter` using method `AddListener`. They have to implement interface `MessageListener`, whose `Receive` method will be called when a message arrives.

Upon creation, an instance of `PullPushAdapter` creates a thread which constantly calls the `Receive` method of the underlying `Transportable` instance (e.g. a channel)[7]. When a message is received, if there is a registered message listener, it will be called (that is, its `Receive` method will be invoked).

As clients are notified when a message is received (vs. actively having to pull messages from a channel), a `PullPushAdapter` eliminates the need for the clients to allocate a separate thread for receiving messages.

3.2.1 Example

The sample code in fig. 3 shows a push-style `PullPushAdapter`. Contrary to the pull-style channel shown in fig. 2, the sample application's `Receive` method will be called whenever a message is received, eliminating the need for a separate message receiver thread. Note that the code uses a `JChannel` instead of an `EnsChannel` in the previous example.

[6] This interface contains only two methods: one for receiving and one for sending messages. Most of the patterns presented here actually need only a reference to a class implementing this interface.

[7] Note that the channel currently has to be connected, otherwise an error message is issued.

```java
public class PullPushTest implements MessageListener {
    private Channel          channel;
    private PullPushAdapter  ad;

    public void Receive(Message msg) {
        System.out.println("Received msg: " + msg);
    }

    public void Start() throws Exception {
        channel=new JChannel("PullPushTest", null); // use default props
        ad=new PullPushAdapter(channel, 1);
        channel.Connect(1000);
        ad.AddListener(this);
        for(int i=0; i < 10; i++) {
            channel.Cast(new String("Hello world").getBytes());
            Thread.currentThread().sleep(1000);
        }
        channel.Disconnect();
        channel.Destroy();
    }

    public static void main(String args[]) {
        PullPushTest t=new PullPushTest();
        try {t.Start();}
        catch(Exception e) {System.err.println(e);}
    }
}
```

Figure 3: `PullPushAdapter` sample

3.3 Synchronous Message Call

A synchronous call object sends an asynchronous message to a group and waits for the first, the first n, or all responses. The caller is blocked until a response is returned. The application program does not need to deal with correlation of response(s) to corresponding request, but this functionality is included in `SyncCall`.

`SyncCall` objects are intended to be used on objects that implement the `Transportable` interface. They emulate synchronous message exchange on top of an asynchronous message transport, that is the sending of a message to a group (or a single member) and the reception of the result in one step. When sending messages to a group, there might be none, one or more responses. The SyncCall interface allows to specify how many responses should be returned (one, none, n, or all). Additionally, a timeout defines the maximum amount of time to wait for the arrival of a message.

To implement a synchronous message call, `SyncCall` is given an object that implements interface `Transportable` (e.g. a channel). This interface contains a `Send` and a `Receive` method, which allows `SyncCall` to send a request and wait for the corresponding response(s). Other patterns such as `RemoteMethodCall` (3.5) make use of `SyncCall`.

3.4 MethodCall

This class represents a local method call. It is created giving the name of the method to be invoked (later) and a number of arguments. Arguments can also be added separately (they have to be added in the order of their formal parameters). Method `Invoke` takes as argument the target object on which the method is to be invoked. It selects (using Java's Core Reflection API [Sun96]) the correct method of the target object according to the target object's type, the number and types of its arguments, invokes it and returns an `Object`,

which is either a return value (can also be null), or an exception. `MethodCall` is extended by `RemoteMethodCall` (3.5) to invoke methods in remote objects.
Finding the correct method to be invoked can be a complex process. The default method resolution mechanism implemented by method `Class.get[declared]Method` follows a minimalistic approach which in certain cases might not produce the desired result.[8] [Ban98] describes a more flexible approach to dynamic method lookup, similar to the method resolution approach used in CLOS [Ste90].

3.5 RemoteMethodCall

`RemoteMethodCall` extends `MethodCall` with the ability to invoke methods in remote objects. Its constructors additionally accept a transport (`Transportable`) (e.g. a channel) over which the method call will be sent to the remote object.
As a remote method call to a process group may return more than a single response, two methods are added which use `SyncCall` (3.3) to send a request to the remote object and return the first or n responses: `SendGetFirst` invokes the remote method in all group members and returns the first response received as an object (or exception), or null, if a timeout occurred. `SendGetN` invokes the remote method in all group members and returns n responses, or null, if a timeout occurred (and no response has been received). If n is 0, no responses are expected, essentially making the remote method call one-way.
`RemoteMethodCall` is used on the client side. Its equivalent on the server side is `MethodInvoker` (3.6).
In its dynamic way of invoking methods of remote objects, `RemoteMethodCall` bears similarity to JEDI [ADMR97]. However, JEDI focuses on unicast communication.

3.6 MethodInvoker

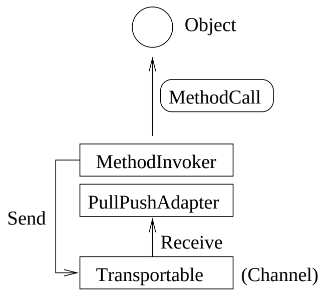

Figure 4: Architecture of `MethodInvoker`

A `MethodInvoker` is used on the server side to invoke methods sent by a client (using `RemoteMethodCall`).
As shown in fig. 4, a method invoker uses a transport to receive method invocations and to send responses. Upon instantiation it creates an instance of `PullPushAdapter` with which it registers. Whenever the `PullPushAdapter` receives a message, it will call the method invoker's `Receive` method. The latter extracts a `MethodCall` object from the message's byte buffer and invokes it against its registered object. When the return value is an exception, it will be thrown, otherwise the return value will be returned to the caller, i.e. the method invoker uses the transport to send the response back to the caller.

[8] Also, dynamic method resolution (at runtime) does not semantically conform to static method lookup (compile-time).

Note that client and server roles may be switched at will, as processes in the server role (using `MethodInvoker`) may themselves become clients (using `RemoteMethodCall`) and clients may become servers at any time (by registering themselves with an instance of `MethodInvoker`). The combination of `MethodInvoker` in the server role and `RemoteMethodCall` in the client role make up for simple and light-weight remote method invocation communication mechanism. However, if more than a single object needs to be registered in a server, or more than one group needs to be joined, and / or client and server roles need to be combined in a single pattern, then class `Dispatcher` (section 3.7) can be used.

3.7 Dispatcher

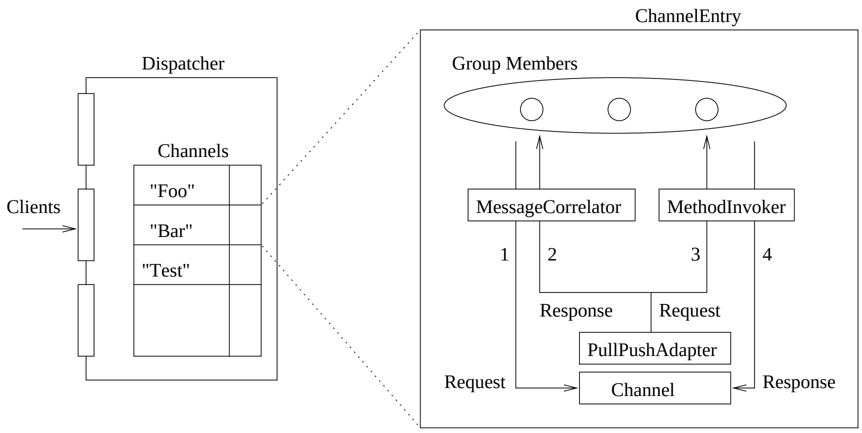

Figure 5: Architecture of Dispatcher

A `Dispatcher` maintains a number of channels and allows clients to join one or more of those channels, and send and receive messages to/from channels. When an object joins a channel (given the channel name), it will receive all messages of that channel as method invocations. Note that in order to receive all messages correctly, an object should implement the methods required by the group (application-specific), e.g. if a member multicasts method `"Foo"`, then all members must contain a method called `"Foo"`. It may itself invoke methods on all members of the channel.

Note that an object that has not previously called Join() is nevertheless able to *send* messages to the group members and receive responses, but requests dispatched to the group members will not be received by it.

The `Dispatcher` class is a replacement for classes `RemoteMethodCall` (client role) and `MethodInvoker` (server role). Instead of using two classes, client-server applications can more conveniently be written using the dispatcher. Its main advantage is that, instead of assuming one client and one server, it allows multiple clients to issue requests and register to get their methods invoked. In one line, the dispatcher is a more sophisticated class offering the combined interfaces of both `RemoteMethodCall` and `MethodInvoker` and allows multiple objects to be registered.

The architecture of the `Dispatcher` is shown in fig. 5. A hashtable maintains a name and a channel entry (`ChannelEntry`) for each channel created in the dispatcher. Clients wishing to send messages have to specify which channel to use (by giving its name).

A client does not have to create a new channel explicitly, but may just call the dispatcher's `Join` method. When the channel already exists, the caller will be added to the channel's object list, otherwise a new channel will be created (`ChannelEntry`) and added to the dispatcher's hashtable with the channel's name as key.

A ChannelEntry consists essentially of a channel, a PullPushAdapter, a message correlator (a pattern not discussed here) and a MethodInvoker. Sending a message using the dispatcher involves the following steps: first the ChannelEntry corresponding to the name given in the call is retrieved. Then a Message is created and registered with the message correlator instance under its message ID. Subsequently the message is sent (1), using the channel as transport. Finally, the result (or results) is retrieved using the message correlator (2).
The previous description applies to a caller in the client role. The server role process is as follows: when a message is received by the PullPushAdapter, it will be forwarded either to the message correlator if it is a response, or to the MethodInvoker if the message is a request.[9] The case of a response was treated above (2), when a request is received (3), the method invoker constructs a MethodCall and applies it to all of the target objects listening to that channel in turn. The response is sent back using the channel (4).

3.7.1 Example

```
public class DispatcherTest {

    public Date GetDate() {return new Date();}

    public static void main(String args[]) {
        DispatcherTest obj1, Object d;
        Dispatcher     disp=new Dispatcher(new JChannelFactory(), null);

        try {
            obj1=new DispatcherTest();
            disp.Join("GroupA", obj1);

            while(true) {
                d=disp.SendGetFirst("GroupA", "GetDate", null, 3000);
                if(d != null)
                    System.out.println("Received response: " + d);
                Thread.currentThread().sleep(2000);
            }
        }
        catch(Exception e) {System.err.println(e);}
    }
}
```

Figure 6: Dispatcher sample

The code demonstrates that an object acting in the server role (offering method GetDate) can at the same time also act in the client role by invoking GetDate on all members of the group and displaying the first result returned.
By creating a dispatcher, we are able to invoke remote methods on all members of a group (in this case "GroupA") and receiving return values. By *joining* a group, we are *additionally* able to act as a server for method GetDate.
It is clearly seen here that the value of a dispatcher lies in the simplicity with which methods can be invoked; providers (servers) of methods do not have to receive messages, find out the correct method to invoke, generate a result and use a transport to send the result back to the caller. Instead, all of this is automatically performed by the Dispatcher class.

[9] Whether a message is a request or a response is determined by a flag in the message itself.

4 Protocol Stack

The protocol stack is JavaGroups' default transport (c.f. fig 1), written entirely in Java. Similar to [PHOA89, VRB95], it uses *micro-protocols* in its implementation. A micro-protocol (from now on called *layer*) enforces a part of the quality of service properties guaranteed by the protocol stack as a whole. The properties desired by the user of a stack are achieved by creating a layer for each property and stacking them on top of each other. Each layer has the same interface (by subclassing a common superclass), which allows to stack any layer on top of any other. However, random stacking of layers will probably not make sense semantically in most cases.

All layers are instances of Java classes and are maintained by a `ProtocolStack` object which itself is connected to an instance of `JChannel` (cf. fig. 1). Adjacent layers are connected by two queues, one for storing messages to be sent down the stack, and the other one for messages traveling up the stack, which guarantees FIFO delivery of messages between layers.

A message sent by `JChannel` is simply passed to the protocol stack, which in turn forwards it to the top-most layer. Each layer performs some computation and then passes the message on to the layer below it. The bottom-most layer typically puts the message on the network.

In the reverse direction, the bottom-most layer of a different protocol stack will receive the message from the network and pass it on to the layer above it. This layer will perform some computation (possibly strip a header from the message) and pass it on to the layer above it. The message travels up the stack until it is received by the protocol stack object, which puts it in a queue for client applications to receive. The message will be removed when `JChannel.Receive` is called.

Messages are simple Java classes and contain a destination- and source address and a byte buffer. *Headers* of arbitrary data can be added to a message and removed again later by the corresponding layer of a different stack, allowing layers to add protocol specific data, such as a checksum or a key for encryption. A message traveling down the stack would typically accumulate a number of headers (possibly one per layer); the corresponding layers would then remove them again from a message traveling up the stack.

The value of layers is that they are self-contained small pieces of functionality, independent from other layers, although, since they are only a small part of the whole stack, they of course depend on other layers to be present as they require their functionality. Since layers are relatively small, they can be verified for correctness more easily than large chunks of (interdependent) code. Also, the concept of layering forces better structuring; as code for a certain type of functionality is localized in one place, layers can be easily replaced/upgraded with new versions.

A protocol layer typically either modifies a message (e.g. by adding a header), or it may delay its delivery, for example to preserve ordering in a FIFO layer in case the message arrived out of sequence.

When it is a protocol layer instead of the `JChannel` object that sends a message, it will in most cases not need to travel up all the way to the `JChannel` object in the receiving stack, but it should probably be caught by the corresponding layer and be processed there. To this purpose, each message has a *layer type tag*, describing the layer from which it originated. A layer generating a message stamps it with its type (all types in a stack have to be unique). Each layer checks whether a message's layer type matches its own. If this is the case, the message will be processed by the layer (and then possibly discarded), otherwise it will just be passed on to the next layer.

The interface of a protocol layer contains methods to process messages from layers above or below (`Down` or `Up`) which will be called when a message travels 'through' that layer. There are also methods to start and stop a layer and to initialize it with data, these are invoked when setting up a new protocol stack or when shutting down an old one (e.g. when a member leaves).

Protocol layers are created by the protocol stack (using a *configurator*) according to a *proper-*

ties argument defined when creating an instance of `JChannel`. A properties string might be `"UDP(mcast_addr=228.2.2.5):NAK:FRAG(size=8096):FIFO:GMS"`. The configurator parses the string, creates the corresponding instances (`"UDP"`, `"NAK"`, `"FRAG"`, `"FIFO"` and `"GMS"`), sets their initial data (`"mcast_addr=228.2.2.5"`, `"size=8096"`) and connects them to each other. The top layer will be connected to the protocol stack object. Then the configurator iterates through the protocol stack and starts each layer in turn. When shutting down a stack, the configurator stops each layer in turn, giving it time to process outstanding messages and then destroys the stack.

5 Conclusion and Future Work

JavaGroups is an early version of a reliable group communication toolkit written entirely in Java. Its major goal is the establishment of a library of frequently used structural and algorithmic patterns to facilitate the development of group applications and protocols. When developing the Java protocol stack, more patterns were identified, merged with existing ones and integrated into the pattern hierarchy. The value of patterns is that they are well-tested, small pieces of recurring software design, making new applications/protocols more robust and reducing development time through reuse.

Some of the patterns make extensive use of Java's advantages. The property of a protocol stack is defined as a string, which results in Java instances being created based on the protocol layers' class names present in the string, drawing on Java's ability to create instances given their class name at runtime. Patterns such as `MethodCall` and `MethodInvoker` use Java's reflection API to dynamically assemble method calls and dispatch them at runtime. Also, we will investigate into dynamically downloading code to a client in *applications* (something that is done today only in applets), e.g. downloading a protocol stack, so that the client does not need to have the protocol classes available when started.

JavaGroups is work in progress and future work will include capturing more patterns, refining existing ones and merging multiple patterns to achieve higher abstraction levels. Most of the patterns discussed in this paper are centered around reliable group RPC-like communication. However, we are more and more focused on behavioral patterns, which capture higher level communication exchanges often encountered in protocol design. An example is distributed commit which can be used e.g. in the implementation of a flush protocol [VRB95]. Another example is a state exchange protocol that can be used to update a joining member with the state of the group, without stopping communication between group members. Higher level patterns could even model primary-backup or coordinator-cohort replication schemes [Bir96, pp. 331–334].

Our work draws from previous research on abstractions/patterns for reliable group communication such as Consul [MS92], Coyote [Coy97], Cactus [Cac98] and BAST [GG97b, GG97a]. All focus to some extent on structuring the development of reliable distributed systems through modularization. Consul provides relatively coarse-grained abstractions, whereas its successors Coyote and Cactus strive to make them more fine-grained. BAST already focuses on patterns for building reliable protocols. It provides patterns of different reliability guarantees, letting the user choose the level of reliability required. Patterns include for example reliable message passing at the lowest reliability level, and at a higher level, consensus and atomic broadcast which inherit the properties of their superclasses.

However, whereas Consul, Coyote and Cactus are based on C++, BAST uses Smalltalk, and none of them is (yet) based on Java. In contrast, our work focuses on Java, striving to integrate reliable group communication seamlessly into the language, exploiting Java's advantages wherever possible. Patterns are the center of our work, which contrasts to other work, which either focuses on patterns, but is not based on Java, or which is based on Java, but does not exclusively focus on patterns.

When a new toolkit for group communication becomes available, it should be straight-forward to integrate it with JavaGroups, so that existing applications based on our collection of patterns can continue to work, using the new toolkit. Note that since most patterns that depend on a message transport only need a function for sending and one for receiving messages (interface `Transportable`), it should be easy to port the patterns to a different group communication toolkit, without needing to port the entire JavaGroups toolkit.

When developing behavioral patterns, one particular interest is how they can be used by clients and servers[10] without having to modify client or server code. We are currently experimenting with code that can be dynamically 'glued' to clients and servers, adding new (protocol) functionality to them without need for modification.

More information about JavaGroups is available at
`http://www.cs.cornell.edu:/home/bba/javagroups.html`.

References

[ADMR97] Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin. *Providing Easier Access to Remote Objects in Distributed Systems.* California Institute of Technology, Pasadena, CA 91125, 1997.

[Ban98] Bela Ban. Static vs. Dynamic Method Resolution in Java: The Case For Argument-Based Method Selection. http://www.cs.cornell.edu/home/bba/papers.html, 1998.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Co., 1996.

[BR94] K. P. Birman and R. Van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[Cac98] Dept. of Computer Science, University of Arizona. *CACTUS: An Integrated Framework for Dynamic Fine-Grain QoS*, 1998. http://www.cs.arizona.edu/cactus/overview.html.

[Coy97] Dept. of Computer Science, University of Arizona. *Coyote: An Approach to Constructing Configurable Fault-Tolerant Distributed Services*, 1997. http://www.cs.arizona.edu/coyote/overview.html.

[GG97a] Benoit Garbinato and Rachid Guerraoui. BAST: A Framework for Reliable Distributed Computing. Technical report, Ecole Polytechnique Federale de Lausanne, 1997.

[GG97b] Benoit Garbinato and Rachid Guerraoui. Using the Strategy Pattern to Compose Reliable Distributed Protocols. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, June 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Hay98] Mark Hayden. The Ensemble System. Technical Report 98-1662, Cornell University, January 1998.

[ibu98] Softwired Inc. *iBus - The Java Multicast Object Bus*, 1998. http://www.softwired-inc.com/ibus.

[10] Clients and servers are roles that may change dynamically, e.g. when a server servicing a request from a client contacts another server, it itself becomes a client for the duration of that request.

[MPS92] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Modularity in the Design and Implementation of Consul. Technical report, CS. Dept. University of Arizona, 1992.

[MS92] Shivakant Mishra and Richard D. Schlichting. Abstractions for Constructing Dependable Distributed Systems. Technical Report TR 92-19, CS Dept. University of Arizona, 1992.

[MS97] Silvano Maffeis and Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, February 1997.

[PHOA89] Larry L. Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the x-Kernel: Evaluating new Design Techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, Arizona, November 1989.

[SC97] Douglas Schmidt and Charles Cranor. Half-Sync/Half-Async. An Architectural Pattern for Efficient and Well-Structured Concurrent I/O. Technical report, University of Illinois at Urbana-Champain, 1997.

[Ste90] Guy L. Steele. *Common LISP. The Language*. Digital Press, 1990.

[Sun96] Sun Microsystems Inc. *Java Core Reflection. API and Specification*, October 1996.

[VRB95] Robbert Van Renesse and Kenneth P. Birman. Protocol Composition in Horus. Technical Report TR95-1505, Cornell University, March 1995. http://www.cs.cornell.edu/Info/Projects/HORUS/Papers.html.

[VRBM96] Robbert Van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus, a Flexible Group Communication System. *Communications of the ACM*, April 1996.