JavaGroups – Group Communication Patterns in Java

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

July 31 1998

1 Overview

JavaGroups is a group communication toolkit written in Java providing reliable multicast communication. Its first version uses the group communication protocols provided by the Ensemble [Hay98] distributed communication toolkit, but future versions will have their own built-in group communication protocols.
Group communication allows communication endpoints to form a *group*. Messages sent to a group are received by all members of that group. New members may join a group (and subsequently receive all messages sent to the group), and current members may leave a group at any time. Members that have crashed will eventually be removed from a group.
Communication endpoints can be processes or objects, essentially any entity that can send and receive messages to/from a group.
At the core of JavaGroups is a low-level abstraction of a communication endpoint of a group – a *channel*. When a channel is created it is given a name. Multiple channels with the same name form a group. A new channel automatically joins the group under the given name, and leaves it when it is destroyed. Groups only exist conceptually and can be referred to by their name. Channels are used to represent a member of a group.
Channels can be used to send messages to single members, a subset of the members, or all members. When a channel receives a message, it is stored until a client dequeues it from the channel. In order to be able to send and receive messages, a client must first connect to the channel. Only one client can be connected to a channel at any time. Other clients can only connect to the channel when the previously connected client disconnects from the channel.[1]

[1] However, other clients may create a new channel with the same name.

Channels are the lowest-level abstraction provided by JavaGroups to create group communication aware applications. Since they are conceptually similar to sockets, their use should be straight-forward to programmers familiar with sockets. However, like sockets, channels provide only a relatively low-level abstraction for asynchronously sending and receiving messages. Any task more complex, like for example synchronous message exchange, RPC-like communication, correlation of request with response(s), or remote method invocation, has to be written by the programmer. One of JavaGroups' goals is to provide small pre-fabricated building blocks (*group communication patterns*) that perform exactly these tasks, on top of the channel, allowing programmers to access group communication at a higher level of abstraction. These building blocks should be (a) sufficiently small, so that they can be used independently, and (b) they should be able to be combined to create larger building blocks.
A channel is an abstract class and needs to be specialized to provide real group communication facilities. In the current version, a subclass (`EnsChannel`) is provided that accesses Ensemble to do this. A future version will contain a native Java implementation of Ensemble's group communication protocols (e.g. `JChannel`).

1.1 Architecture

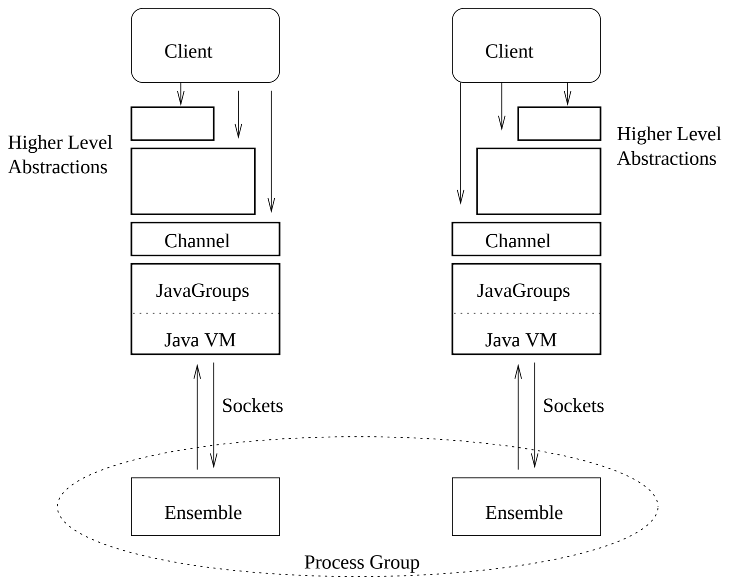

Figure 1: Overview of Architecture

The current architecture (using Ensemble) is shown in fig. 1. Clients may access one of the higher-level abstractions, or they may access channels directly. Note that since `Channel` is an abstract class, a client will always have to access a specific subclass of `Channel`, e.g. `EnsChannel` for a channel implementation that uses Ensemble. The (Java) implementation of `EnsChannel` starts up an Ensemble process with which it communicates via sockets. Ensemble takes care of finding other Ensemble process groups with the same name and joining them.[2]

1.2 Communication Modes

A number of communication modes, as described below, can be identified for group communication. When designing interfaces for group communication, these (sometimes conflicting) modes have to be taken into account.

1.2.1 Message Exchange: Synchronous vs. Asynchronous

Communication between a sender and one or more receivers can be synchronous, that is the caller is blocked until one or all responses are returned, or asynchronous where the caller returns immediately after sending the message. In the latter case, the caller has to retrieve the response(s) later and correlate it/them to the request sent (usually using some form of message numbering).

1.2.2 Message Reception: Pull vs. Push

Pull-style message retrieval means that the caller actively has to retrieve the response(s) after sending a request while with the push-style the caller will be sent any responses. In the first case, the caller may have to allocate a thread to retrieve the responses in the background, in order to be able to continue the work. In the second case, usually a callback (a function or method) will be invoked when a response is received (or, alternatively, when all responses have been received).

1.2.3 Message Sending vs. Remote Method Invocation

The content of data exchanged between entities can be a low-level message containing sender and receiver address, a message number and a simple byte buffer, or it can be on the higher abstraction level of a method invocation on remote objects. While the latter seeks to provide transparency by extending the procedure call paradigm to remote group objects (RPC), the former would typically be used when sending unstructured data (e.g. audio, video) over a channel.
Note that remote method invocation is mostly synchronous while message sending can be synchronous or asynchronous.

[2]Note that the Ensemble executable (outboard) has to be available. Also, there has to be a gossip process started if IP multicasting is not available. Refer to [Hay98] for details.

1.2.4 1 Response vs. N Responses

When sending a request to a group, there might be none, one or more responses, depending on the policy chosen. Sometimes just the first response received is needed (e.g. in actively replicated servers), while at other times all responses are needed (e.g. results of a computation, load-balanced over all members of a group).

1.2.5 Timeout vs. No Timeout

All modes of communication may be accompanied with a timeout which is supposed to terminate blocking calls after a certain time span has elapsed. Timeouts allow to determine the maximum amount of time needed for certain calls.

2 Group Communication Patterns

Group communication patterns are *building blocks* and / or *communication algorithms* frequently encountered in applications employing group communication. Some patterns make use of (the abstract notion of) channels, while others use channels only indirectly and / or can be used independently altogether. They greatly simplify programming by encapsulating recurring pieces of group communication into sufficiently small and reusable Java classes. There are three types of patterns: following the terminology in [GHJV95] they are *creational*, *structural* and *behavioral patterns*. Creational patterns deal – as their name implies – with creation of components (either structural or behavioral ones), abstracting the instantiation process. Structural patterns are concerned with how objects and classes are composed to form larger structures. They typically show in the form of classes. Behavioral patterns are concerned with capturing algorithms in a reusable form. They are typically represented by class methods.
A programmer is free to choose whether to use channels and none of the patterns, or whether he requires a higher level of abstraction by using one (or more) patterns.
An example of a creational pattern is the `Configurator` class which creates, initializes and starts a protocol stack given a setup string.
An example of a structural pattern is the `SyncCall` class which emulates synchronous message exchange on top of (inherently asychnronous) channels. Its value is that the caller does not have to correlate one or more responses to requests himself, but is blocked until the first (or any N responses) is returned.
An example of a behavioral pattern is failure detector which regularly pings designated members of a group and – if a member crashed – indicates this to the group membership service.[3]
Both structural and behavioral patterns are intended to be used in both developing communication applications and protocols.

[3]This service has not yet been implemented.

The value of patterns is that they are independently reusable pieces of recurring design decisions which are based on a very generalized concept of group communication (namely channels) and can therefore also be used on top of other group communication toolkits.[4] However, contrary to frameworks, which impose a certain structure of usage on clients, JavaGroups is only a toolkit which gives programmers the freedom to decide which pieces to use and which not.

2.1 Utility Classes

2.1.1 DistributedHashtable

A `DistributedHashtable` is derived from `java.util.Hashtable` and allows to create several instances of hashtables at different locations. All of these instances have exactly the same state at all times. When creating such an instance, a group name determines which group of hashtables will be joined. The new instance will then query the state from existing members and update itself before starting to service requests. If there are no existing members, it will simply start with an empty state.

Modifications such as `put`, `clear` or `remove` will be propagated in orderly fashion to all replicas. Read-only requests such as `get` will only be sent to the local copy. Since both keys and values of a hashtable will be sent across the network as copies, both of them have to be serializable, i.e. implement interface `Serializable`. This allows for example to register remote RMI objects with any local instance of a hashtable, which can subsequently be looked up by another process which can then invoke remote methods (remote RMI objects are serializable). Thus, a distributed naming and registration service can be built in just a couple of lines.

A `DistributedHashtable` allows to register for notifications, e.g. when a new item is set, or an existing one removed. All registered listeners will notified when such an event occurs. Notification is always a local process; for example in the case of removing an element, first the element is removed in all replicas, which then notify their listener(s) of the removal (after the fact).

2.1.2 Messages

A `Message` is a simple class containing a unique ID, the receiver's and sender's address (as an `Object`), a byte buffer to contain the actual data and a flag indicating whether it is a request or a response.

It will be extended in that there will be additional methods for storing and retrieving objects and atomic Java types to / from the byte buffer, for adding data at the head of the buffer (without copying, see 2.1.3) etc.

[4]Note that those patterns that require a channel most often only require a SEND and RECEIVE Java interface. Patterns that are not directly layered on top of channels are named *utility patterns*.

2.1.3 Data

Class `Data` maintains a byte buffer and adds methods to add data (raw bytes, atomic Java types or objects) at the head or tail of the byte buffer. Also, it allows to read / write objects to / from the buffer. `Data` is used by class `Message` (2.1.2).

2.1.4 Sets

Containers for objects (e.g. addresses or endpoints). Includes methods for adding, removing members, and intersection / union with other sets. *Not yet implemented* See `JavaGroups.JavaStack.Membership`.

2.1.5 Queue

The `Queue` class is widely used in producer – consumer communication. A producer puts items in the queue, and a consumer removes them from the queue. When there are no items available, a consumer is blocked (unless it specified a maximum timeout to wait for items to become available). Addition and removal of items is synchronized, i.e. no two entities (e.g. consumer and producer, or one producer and another producer) can be in the queue at the same time.

2.2 Creational Patterns

2.2.1 Configurator

Only preliminary, describe in more detail
Given a *setup string* for a protocol stack such as
`"UDP(port=5674):FRAG(size=1020):NAK:FIFO:TOTAL"`, the configurator treats each part of the string between colons as class name followed by an optional configuration parameter. It creates an instance of each class, starting with `UDP` and layers the classes above each other. Then all the instances are initialized and started.

2.3 Structural Patterns

2.3.1 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 in semantics 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 has first to connnect to it. When done, it should disconnect from the channel. There can always only be a single client connected to the same channel. 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 other 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. It returns the addresses of all members in the group. Since these addresses will typically vary in their form and content, they are only returned in the most general form of `Object`. Each subclass of `Channel` 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 can essentially only receive addresses as result of method calls and subsequently use them as target addresses in sending messages to a single 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 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 do when receiving data from a socket, and second, by not having to use callback into user code at this level, channels cannot get blocked by user-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 channel can be used (such as `PullPushAdapter` (2.3.6) or `Dispatcher` (2.3.10)).

EnsChannel `EnsChannel` is a subclass of `Channel` and a concrete implementation of the latter's methods. To create an instance of it, its constructor accepts a channel name and properties (specified as string). The channel name is used by Ensemble to find all other members (channels) in the process group, and the property string defines the protocol stack that Ensemble is supposed to create for this process group (see [Hay98, CS 97]).

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

Example The code below shows a simple example of how an `EnsChannel` is used:

```
public class ChannelTest implements Runnable {
    private Channel channel=null;
    private String  props="Top:Heal:Switch:Leave:Inter:Intra:Elect:"+
                          "Merge:Sync:Suspect:Top_appl:Pt2ptw:"+
                          "Pt2pt:Frag:Stable:Mnak:Bottom";

    public void Start() throws Exception {
        channel=new EnsChannel("TestChannel", props);
        channel.Connect(3000);
        new Thread(this, "ChannelTestThread").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 {
            ChannelTest test=new ChannelTest();
            test.Start();
        }
        catch(Exception e) {System.err.println(e);}
    }
}
```

First a new channels is created given a name and certain properties (used by Ensemble). Then the channel is connected (a wait of 3 seconds is added to let the channel discover other channels of the same name). The main part of the `Start` method sends 10 messages to all channels with the same name (i.e., the process group), waiting 1 second between sends, and finally disconnects from the channel and destroys it.
Note that a separate thread is started to perform the task of receiving messages from the channel (remember, channels use the pull-style, so message have to be

retrieved actively). When a message is received, its contents are converted into a string and printed on stdout.
For an example demonstrating the push-style see section 2.3.6.

2.3.2 Synchronous Group Message Call (`SyncCall`)

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 response(s) is/are returned.
SyncCall objects are intended to be used on objects that implement the `Transportable` interface. They allow to 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).
`SyncCall` does not use the `MessageCorrelator` class, but instead implements a simple(r) and less sophisticated message correlation scheme.
Other patterns such as `RemoteMethodCall` (2.3.8) make use of `SyncCall`.

2.3.3 Message Correlator

This class correlates requests with their responses. The caller can choose to get the first response, N responses or all responses (each call can also include a timeout to prevent having to wait forever).
It offers essentially 4 major methods: `AddRequest` to insert into its hashtable a message keyed by its message ID, `AddResponse` to add a response macthing the request message ID to the corresponding item's queue in the hashtable, `GetResponse` to wait until either a response was received or a timeout has occurred and `GetResponses` to wait until N responses have been received or a timeout has occurred.
The message correlator is for example used by the `Dispatcher` (2.3.10).

2.3.4 MUX

Maintains several channels, caller specifies channel over which message is to be sent. *Not yet implemented. Is it used at all ? Dispatcher does the same ...*

2.3.5 Demux

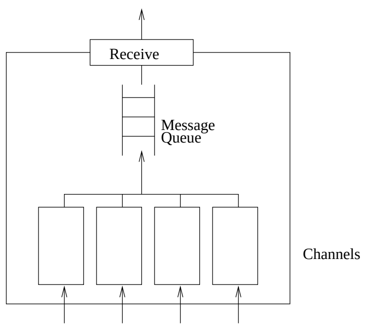

Figure 2: Architecture of Demultiplexer

A demultiplexer combines the output of multiple channels into a single (virtual) channel. Channels can be added and removed from the demultiplexer. For each new channel, a separate thread is started which retrieves messages from the channel and adds them to a message queue maintained by the multiplexer. Clients calling method `Receive` will receive the message retrieved from that queue. Note that this pattern is similar to the Unix SELECT system call.
Although the `Demux` class implements interface `Transportable`, it is only envisaged that `Receive` should be used. Method `Send` is not implemented; for sending messages, the underlying transport should be used (e.g. a channel).

2.3.6 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 function or method which is invoked whenever a message has been received. It allows a client of a channel to be notified when messages have been received instead of having to actively poll the channel for new messages. This eliminates any need for the clients to allocate a separate thread for receiving messages.
A `PullPushAdapter` is always created with a reference to a class that implements interface `Transportable` (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. Any number of clients can register and all of their `Receive` methods 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 chan-

nel)[6]. When a message is received, if there are any registered message listeners, all of them will be called (that is, method `Receive` will be invoked) in turn.
As this class does not implement interface `Transportable`, but uses it for receiving messages, an underlying object has to be used to send messages (e.g. the channel on top of which an object of this class resides).

Example

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

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

    public void Start() throws Exception {
        channel=new EnsChannel("PullPushTest", null);
        conv=new PullPushAdapter(channel, 1);
        channel.Connect(1000);
        conv.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);}
    }
}
```

Contrary to using channels (using pull-style message reception), here no separate thread has to be allocated to receive message. Instead, a `PullPushAdapter` is layered on top of the channel and a reference to the client object added. This causes the client's `Receive` method to be called whenever a message has been received by the `PullPushAdapter`. Note that compared to the pull-style example, push-style message reception is considerably easier (no separate thread management) and requires less code to program.
For an example showing the use of the pull-style see section 2.3.1.

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

2.3.7 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 returns an `Object`, which is either a return value (can also be null), or an exception. `MethodCall` is extended by `RemoteMethodCall` (2.3.8) to invoke methods in remote objects.
Finding the correct method to invoke 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.[7] [Ban98] describes a more flexible approach to dynamic method lookup, similar to CLOS' method resolution approach.
***** Add section on types of parameters, e.g. no atomic types !!**
Describe: Automatic mapping of object arguments to atomic types, also mapping of atomic return values to objects.
***** Add section on matching of arguments with formal parameters**

2.3.8 RemoteMethodCall

`RemoteMethodCall` extends `MethodCall` (2.3.7) 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` (2.3.2) 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 mainly used on the client side. Its equivalent on the server side is `MethodInvoker` (2.3.9).
In its dynamic way of invoking methods of remote objects, `RemoteMethodCall` bears similarity to JEDI [ADMR97]. However, JEDI focuses on unicast communication.

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

2.3.9 MethodInvoker

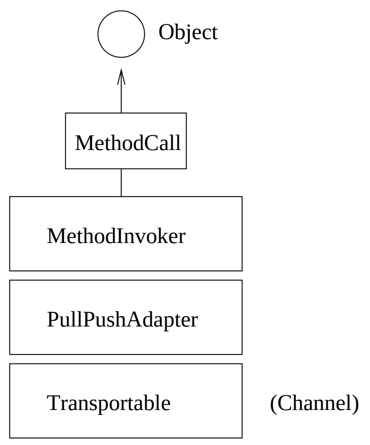

Figure 3: `MethodInvoker` example

A `MethodInvoker` is used on the server side to invoke methods sent by a client (using `RemoteMethodCall`).
As shown in fig. 3, a method invoker uses a transport to receive method invocations and to send responses. When created, it automatically creates an instance of `PullPushAdapter` with which it registers. Whenever the `PullPushAdapter` receives a message, it will the `Receive` method of the method invoker. 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.
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 client 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, if more than one group needs to be joined, and / or if client and server roles need to be combined in a single pattern, then class `Dispatcher` (2.3.10) should be preferred.

2.3.10 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

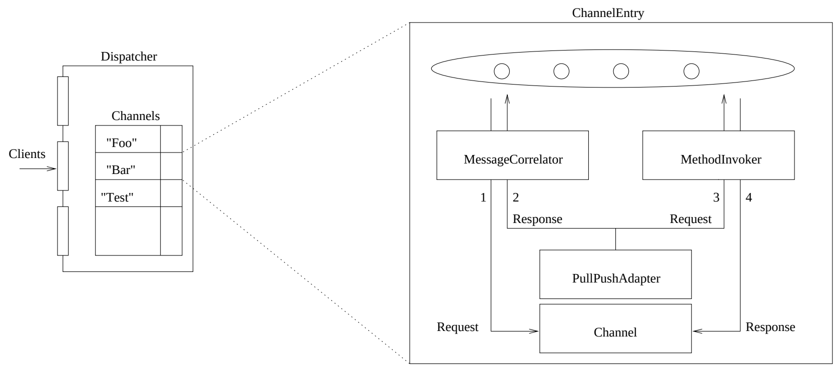

Figure 4: Architecture of Dispatcher

object joins a channel (given the channel name), it will be dispatched all messages received on that channel in the form of method invocations. Note that in order to receive all messages, an object should implement the methods required by the group (application-specific). It may itself invoke methods on all members of the channel (a channel is nothing else than a group !).

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 dispatched to it.

The `Dispatcher` class is a replacement for classes `RemoteMethodCall` (client side) and `MethodInvoker` (server side). Instead of using 2 classes, client-server applications can more conveniently be written using the dispatcher. Its main advantage is that, instead of assuming 1 client and 1 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. 4. 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 explicitly have to create a new channel, may 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 essentialy of a channel, a `PullPushAdapter`, a `MessageCorrelator` and a `MethodInvoker`. Sending a message using the dispatcher involves the following steps: first `ChannelEntry` corresponding to the

name given in the call is received. Then a `Message` is created and registered with the `MessageCorrelator` 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 `MessageCorrelator` (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 is will be forwarded either to the `MessageCorrelator` if it is a response, or to the `MethodInvoker` if the message is a request.[8] 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 return values are sent back using the channel (4).

Example The example code below does essentially the same as the examples in sections 2.3.1 and 2.3.6.

```
public class DispatcherTestChannel {

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

    public static void main(String args[]) {
        DispatcherTestChannel obj1;
        Dispatcher            disp=new Dispatcher();

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

            while(true) {
                System.out.println("Sending method");
                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);}
    }
}
```

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 member dispatchers of the group and displaying the first returned result.
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`.

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

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.

2.3.11 Promise

Promises [LS88] are used to start a computation and return while the computation is being performed (just after starting it). In the context of group communication, a promise is used to send a request to all (or N) members of the group and return immediately. The promise can be checked for completion and number of responses received so far. The caller might perform some other work, periodically check the status of the computation, and – depending on it – retrieve the result(s) from the promise.
Promises allow clients to asynchronously invoke a method call without having to be blocked until the result is returned.
Their main value lies in starting multiple invocations in parallel, and gathering the results later (load balancing).
In contrary to the work described in [LS88], promises as used in the context of JavaGroups are not returned from some method call, but used as starting point as well.

2.3.12 RepeatedUnicast

Frequently clients accessing a server will find that the server has crashed, or for some other reasons, does not respon within a certain time frame. In this case, the client may access another server, possibly at a different location. The `RepeaterUnicast` class allows to specify a number of server locations (addresses) that will be tried out in turn until a response is received, or a timeout has occurred.
An instance of this class is initialized with a transport which will be used to send and receive messages, and a number of addresses to which the request will be sent.
Method `Send` tries to send a `Message` to the first member. If a response is received within `timeout` milliseconds, it is returned. Otherwise, the member is removed and the next address is tried out. If no member address is found to be functional, an exception is thrown.
Method `SendMethod` does the same as `Send`, but instead of just sending a message, a method is sent (in the form of `RemotemethodCall`. The result is returned as an object.
This class is used for example in the group membership protocol (GMS) implementation: a new member keeps sending a JOIN request to each member in succession, until a JOIN is successful (which usually happens at the first attempt).

2.3.13 StableStorage

Kind of an interceptor that - before passing messages on - stores them on stable storage. This allows clients that reconnect after a certain time, to request older messages to be replayed. Could be used in conjunction with `PullPushConverter`: when no client is connected, whenever a message is received, it will be stored. When a client connects, all stored messages are *pushed* to the client and subsequently purged from stable storage.
Not yet implemented

2.3.14 LazyEvaluator

Pushes messages (or method invocations, if used in conjunctions with `MethodCall`) to client(s). Used e.g. with `PullPushConverter`. Rather than blocking the 'pusher', it enqueues messages and uses its own thread to push them one after the other to clients. Thus clients can block the `LazyEvaluator`, but never the 'real' pusher. Allows the pusher to send non time critical notifications to clients without blocking.
Needs to be refined

2.4 Behavioral Patterns

Behavioral patterns are less fine-grained than structural patterns. They capture frequently used algorithms and / or communication exchanges between entities in the domain of group communication. Using structural patterns to accomplish their tasks, they are intended to significantly reduce the amount of code needed to implement protocols and applications. Also, assuming that these patterns have been tested rigorously to assure their correctness, the probability of introducing errors is greatly reduced.
Potential candidates are:

- Group Membership Protocol
- Failure Detector: monitors a group of objects by sending them periodic heartbeats. Notifies registered entities of suspected objects.
- Primary – Backup
- Load Balancing
- Voting and Election Protocols (coordinator election)
- 2-Phase Commit / Distributed Commit
- Flush Protocol

- State Transfer
- Repeating Unicast: continues sending a message to a (unicast) member of a group until a response is received. If a response is not received after some timeout, the next member is used. Continues as long as members are available or a reponse is received. A user callback will be invoked when a member is not available (e.g. to suspect that member and remove it from the group). Used e.g. for JOIN/LEAVE implementation; a new member repeats sending JOIN requests to a member of the group until it receives a response.
- Random Unicast: similar to Repeating Unicast. A unicast message is sent to a randomly selected member of a group. Used for load-balancing.

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. Technical report, Cornell University Computer Science Dept., 1998. `http://www.cs.cornell.edu/home/bba/papers.html`.

[CS 97] CS Dept Cornell University, Ithaca NY 14850. *The Ensemble Distributed Communication System*, 1997. http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html.

[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.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7), July 1988.

[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.