Implementing Group Protocols Using Dynamic Remote Method Calls

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

Abstract

This paper discusses the merits of synchronous group remote method call (G-RMC) to implement peer protocols. Peer protocols are used in layered protocol stacks to communicate horizontally between two layers of the same type. We will show that the higher abstraction level of synchronous RMC leads to less code, better understanding of a peer protocol and improved maintainability. However, ordering properties frequently required by group communication stacks may interfere with the synchronous calling of remote methods, causing deadlocks. We will present a solution that detects and solves deadlocks in the JavaGroups toolkit.

1 Introduction

Protocol layers often have to communicate with their peer layers in different stacks. For example, a group membership service (GMS) protocol layer may want to contact all other GMS layers in the group to solicit a vote for a new coordinator, or to install a new view.
This paper presents a mechanism that allows to invoke remote methods in one or more group members and return the result(s) synchronously. This allows to define protocols like regular classes in object-oriented programming: a protocol layer would be an instance of a class, containing a number of methods. It may make remote method invocations on other instances of the same class, and might itself be invoked by other peers. Seeing peer interaction as remote method invocation leads to a higher level of abstraction, which helps understanding a protocol, leads to fewer lines of code needed for implementation and increases maintainability.
The novelty of our approach is that (a) it is based on remote *method calls* rather than procedure calls, (b) method invocations are *dynamic*, i.e. assembled and dispatched at run-time vs. using client stubs/server skeletons generated at compile-time, and (c) we focus on the deadlock problem arising when synchronous method calls collide with strong ordering properties of the stack.
The mechanism discussed in the paper is used in the JavaGroups project [Ban98], mainly to implement peer protocols. However, it can also be used to implement applications.
We will start out by shortly describing the JavaGroups toolkit, which is the group communication substrate in the context of which the synchronous group RMC was developed. The main section will focus on the building blocks used to implement RMC. Then, we will give an example of a protocol layer that uses RMC to communicate with its peers. Finally, we will look into the problem of deadlocks associated with synchronous RMC and message ordering and offer a possible solution.

2 JavaGroups Protocol Stack Architecture

JavaGroups is a group communication toolkit based on a layered protocol stack architecture similar to [Rit84, PHOA89, VRBM96, CS 97]. An application uses a *channel* as group endpoint. A channel has a local address and a group address which is the name of the group. All channels with the same group address form a group. Some of the methods a channel offers are for connecting to/disconnecting from a group (i.e. joining and leaving), sending a *message* to one or more members, receiving messages, and receiving notifications when members join or leave the group. A message consists of a receiver and a sender field, which are objects denoting the destination and source address. If the destination address is null, the message is sent to all members in the group. Additional fields are the payload (byte buffer) and a *header list*. The former is used to carry information in message, it could contain for example a serialized object. The latter is used by the protocol layers to add information pertaining to their functionality to a message. For example, a key exchange protocol layer may want to add its public key. Headers allow to attach information to a message without altering it.

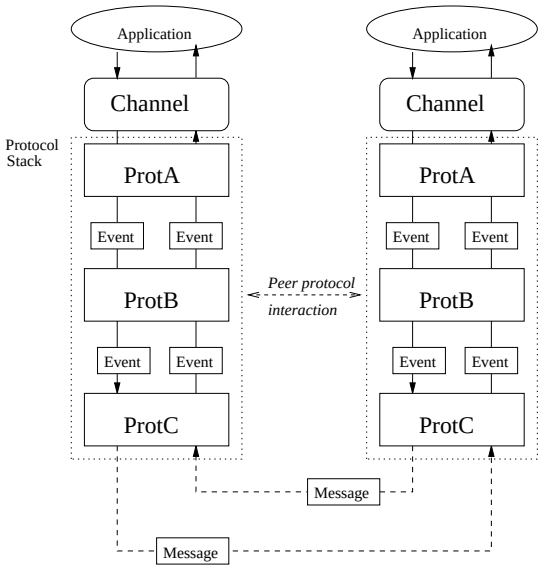

Figure 1: Architecture of JavaGroups

A channel has a reference to a protocol stack object which it uses to send/receive messages. A protocol stack consists of a number of layers, each of which performs a different task, e.g. FIFO ordering, encryption, group membership etc. Data is sent through a stack in the form of *events*. An event has a type flag and an argument. When a message is sent down the stack by the channel, it is wrapped into an event, and tagged as *message* with the message as argument. The bottom layer unwraps the message and puts it on the network. When a message is received, it is again wrapped into an event and passed up the stack. Note that *messages* are exchanged *between* different stacks, whereas *events* are only used to transport data *within* a protocol stack. There are a number of different events such as view change, message, connection/disconnection and suspicion events.

A protocol layer is always (directly or indirectly) derived from class `Protocol`. Depending on its functionality, it has to override a number of methods, the most basic ones being `Up(Event evt)` and `Down(Event evt)`. The former is called when an event is received from the layer below, the latter when an event is received from the layer above. The default implementation is to forward the event unchanged.

The protocol stack is created according the a specification given by the creator of a channel. The specification is a string of the form `"P1:P2:Pn"`, with layers specified from bottom to top

(plus optional parameters), where each string maps to a Java class representing that protocol layer. Each layer is connected to its neighbor by two FIFO queues and the `Protocol` class takes care of adding/removing events to/from queues and calling the `Up` or `Down` methods respectively.
Protocols can be categorized according to their interaction with other protocols: some protocols interact primarily with their neighbor protocols *in the same stack*, and others interact with their corresponding peer instances in different protocol stacks. The fromer type of interaction can be called *vertical interaction*, and typically consists of the following activities:

- pass a message to its neighbor (up or down the stack) unchanged,
- add a header to the message, but not modify it,
- modify the message, add a header and pass it on (e.g. encryption, fragmentation layers),
- reorder the message (e.g. FIFO, TOTAL layers) or
- discard the message (e.g. DUPL layer)

These tasks can easily be done with protocol layers inheriting from `Protocol`.
The second types of protocols interact not primarily with other layers in the same stack, but with their equivalent peer instances *in different stacks*, as shown in fig. 1 between the `ProtB` layers. Therefore this type of *peer protocol* interaction is called *horizontal interaction*. A typical peer protocol is a group membership service (GMS) layer which needs to interact with the GMSs in all member stacks to e.g. install a new view.
For a more detailed description of the JavaGroups protocol stack refer to [Ban98].

3 Building Blocks for Peer Protocols

The philosophy of JavaGroups is to provide only a small and primitive socket-like API (a channel) for group communication, arguing that it is difficult to design an API that fits the needs of all group applications. But similar to higher-level abstractions on top of sockets (e.g. object request brokers), JavaGroups offers *building blocks* that hide the use of channels from the application and offers a more sophisticated interface to the application.
G-RMC is a building blocks built in turn from finer-grained and more general building blocks. We'll describe the `RequestCorrelator` which is used by the `GroupRequest` class to implement G-RMC. Then we'll present two extensions of the `Protocol` class, `MessageProtocol` and `RpcProtocol` which make use of `GroupRequest`. The former allows to send requests (in the form of messages) to a number of peers and synchronously receive responses, while the latter uses the former to implement remote *method invocations*. `RpcProtocol` would have to be chosen as parent protocol to implement a peer protocol based on G-RMC.

3.1 RequestCorrelator

In distributed peer computing (as is the case in groups), any member may be the *originator and receiver* of a request. Therefore, unlike traditional client–server programming, where roles are fixed, a peer process switches roles dynamically. It may even be possible that a process is a client *and* server at the same time, both sending a request and serving another one (in diffent threads).
That means that – besides sending requests and receiving the corresponding responses – every member must be enabled to process requests sent by other members and send responses. The main roles of a peer protocol are therefore

Send requests Client role. Send a request (message) to a single or all group members and receive 0 or more responses. If the request is asynchronous, no responses will be received. If it is synchronous (i.e., the caller is blocked until the call returns, it should be possible to specify how many responses are to be received, e.g. all, majority, first.

Receive responses Client role. Receive a response to a request. If the request was sent to the whole group, receive a number of responses. No response will be received if the request was sent asynchronously. If a member has crashed before sending a response in the synchronous case, its response should be marked as 'not received'. Note that correlation between requests and responses is important, i.e., if a sender sends requests r1, r2 and r3, incoming responses have to be matched against their originating requests.

Receive request Server role. Receive a request sent by a group member, process it and send a response (see below). Since local messages are enabled on a channel by default, a sender will receive its own request and has to process it like any other group member.

Send response Server role. Send a response to a request. Usually invoked when done processing a request.

The above functionality has been incorporated in class `RequestCorrelator`. This class can be used in an application on top of a channel, or to implement a peer protocol.
The structure of a request correlator is shown in fig. 2.

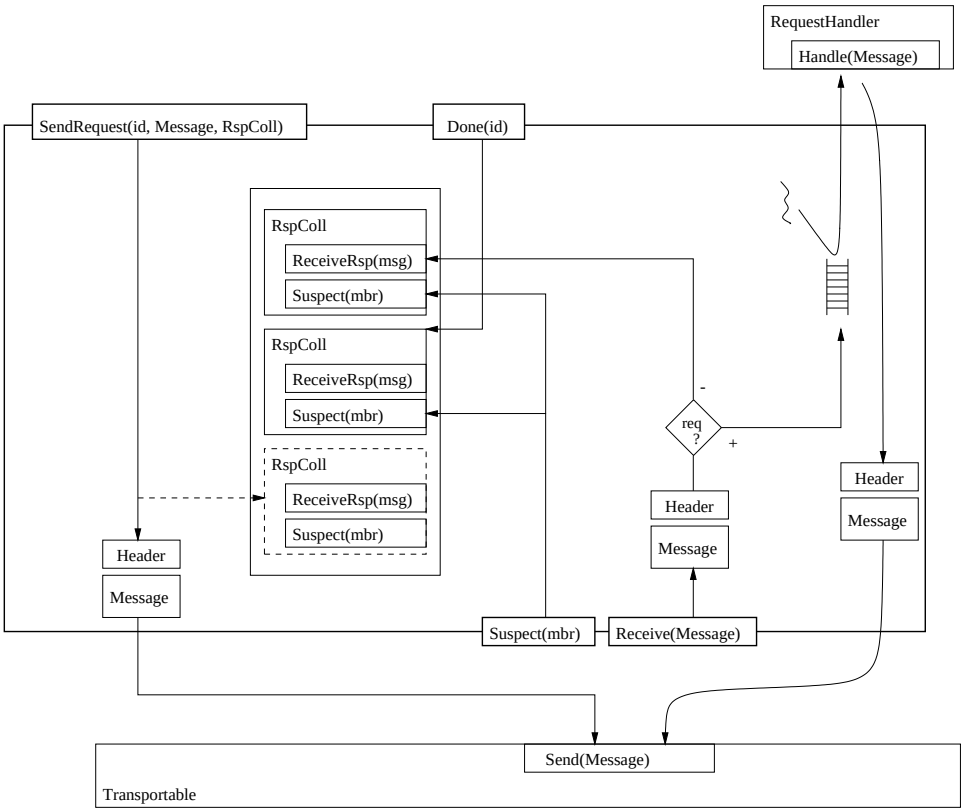

Figure 2: Request correlator

Essentially, a `RequestCorrelator` has 1 object that is responsible for handling all requests (`RequestHandler`), and 0 or more *response handlers* (`RspCollector`) which will receive the response(s) for a specific request. Each request is identified by a unique ID (for example the

current time in milliseconds) and added to a request table. When a response comes in, its ID is matched against the requests, the correct request is found and the response directed to it. Having multiple requests pending at any time (compared to just a single one) gives clients of request correlator more freedom: the decision whether to send a single or multiple requests at a time is theirs.
The `RequestCorrelator` has methods to a) send a request (synchronously or asynchronously, to a single member or to all group members), b) submit a suspicion message, c) make it receive a message and d) remove a request (calling method `Done` with the request ID). It allows to register a request handler which will handle all requests directed towards this request correlator.
The following actions take place when a group request is sent:

3.1.1 Sending a request

The caller creates a new instance of `RspCollector` and invokes `SendRequest` with a unique ID that it wants to assign to the request, a message and the response collector. All the messages received for that specific request will be directed towards the response collector. If it is null, this means that the request is sent one-way, and no responses are to be returned. If it is not null, it is added to a response collector table.
`SendRequest` adds a specific header to the message and uses the underlying transport's `Send` method to send the message to its destination(s). The header contains

1. The ID assigned to the request by the caller
2. The type. A type can only be a request or a response.
3. Whether a response is expected or not. This is determined by `SendRequest`: if a non-null response collector is given as argument, this field will be true, otherwise false. The receiver of a request can thus determine whether to return a response.
4. A name that is associated with this specific instance of request correlator. Every request correlator has a unique name. When removing the header from a message, it is checked whether the header is of type `RequestCorrelatorHeader` and whether the current request correlator's name matches the one in the header. Only if both conditions are true, the message is processed. Otherwise, it is rejected (`Receive` returns false).[1]

The message's destination field determines whether a unicast or multicast message will be sent, i.e. whether the message will be sent to a single member (destination field contains a member address) or to all group members (destination field is null).

3.1.2 Receiving a request

The user of a request correlator has to take care to receive regular and suspicion messages, and to 'feed' them to the request correlator. When a regular message is received, method `Receive` is called. It peeks at the header and checks for 2 things: a) whether the header is of the correct type (`RequestCorrelatorHeader`) and, if yes, b) whether the name field in the header matches the request correlator's name. If either condition is false, `Receive` will return false. A user will thus typically try to feed an incoming message to the request correlator, and, if `Receive` returns false, pass it on to some other layer, e.g. the layer above the current one (when used in a protocol).
If the condition is true, the header type is checked. If it is a request, the message will be added to the request queue (without yet removing the header), where a separate thread will handle it (see 3.1.3). If the header type is a response, the header will be removed from

[1] This is needed to differentiate between request correlators in different protocol layers of the same stack.

the message and the message added to the response collector that matches the header's ID (calling `ReceiveResponse`, see 3.1.4).
When the client feeds a suspicion message to the request correlator, the `Suspect` method callbacks of all request collectors will be called in turn. This allows building blocks on top of request correlators to return gracefully. For example, a group request building block (see 3.2) might block indefinitely when trying to send a request to 3 group members and waiting for all responses. If one of the group members crashes, no response from this member would ever be received. Receiving a suspicion notification, however, allows the group request to terminate the group call (if it received responses from the 2 other members)[2].

3.1.3 Handling a request

When a request is received, it is handled by a separate thread. This thread is only created if a request handler (`RequestHandler`) was given at the time of the request correlator's creation (parameter of the constructor of RequestCorrelator). Otherwise, if requests are received, they will just be discarded if no request handler was installed.
The thread removes the message header (which was not removed by `RequestCorrelator.Receive` before !) and determines whether the message is one-way or not, i.e. whether it needs to send a response. It then invokes the request handler's `Handle` method, which processes the request by interpreting the message's buffer (e.g. extracting an object from it) and returns an object. If a response need to be returned to the sender of the message, then the thread creates a response message, inserts the object into the response's buffer, and adds another header to the message. This header would indicate that the message is in fact a response, and would contain the request's ID. Finally the message is sent using the underlying transport's `Send` method.

3.1.4 Receiving a response

When `RspCollector.ReceiveResponse` is called, the client of request correlator (the same who created the `RspCollector` object) decides what to do. In the case of `GroupRequest`, it might wait until all the responses have been received (minus the responses from suspected members), until the first response has been received, or it might wait for a majority of all responses to arrive. Note that no response will be received if the response collector argument to `SendRequest` was null; no response collector entry will ever be created in this case, and the message will be sent one-way.
When all the responses have been received, method `RequestCorrelator.Done` should be called to remove the response collector entry from the table. If this was not done, response collector entries would accumulate in the table and take up space. `Done` *has* to be called by the request correlator client, because only it knows when 'all' responses have been received: for example 'all' could mean a majority, and in other cases it could mean 'responses from all members'.

3.2 GroupRequest

Although `RequestCorrelator` can be used by itself, `GroupRequest` is a building block making use of the former, that simplifies sending messages to all members of a group and collecting responses. Its most important public methods are:

```
public class GroupRequest {
    public GroupRequest(Message m, RequestCorrelator corr, Vector members,
                        int rsp_mode);
    public GroupRequest(Message m, RequestCorrelator corr, Vector members,
                            int rsp_mode, long timeout, int expected_mbrs);
```

[2]If no suspicion service is available, timeouts can be used instead.

```
    public synchronized boolean Execute();
    public void                 Reset(Vector members);
    public RspList              GetResults();
    public Vector               GetSuspects();
}
```

The 2 constructors need a request correlator, the message to be sent, a response mode, an initial membership, a timeout and the number of expected members.
The initial membership is a vector of member addresses from which responses are expected. If the message's destination is not null, meaning that the message is to be sent to a single destination (unicast), then the membership might only contain a single address, namely that of the destination from which a response is expected. If no response is expected, the membership vector can be empty.
The timeout can be used in combination with, or as replacement for, a suspicion service. If a message is sent to members P, Q and R, and responses are expected from all members, then, if R crashes, a suspicion service will feed this information to the underlying request correlator, which in turn will feed it to the group request object sitting on top of the correlator. A group request that has already received responses from P and Q will block until it receives the response from R, or until a suspicion message about R is received. If no suspicion service is available, a timeout specifies the maximum amount of time a group request should wait for all responses, preventing blocking. Timeouts and a suspicion service can also be combined. A timeout smaller than or equal to 0 means wait indefinitely.
The number of expected members is the number of responses expected. It is only used if the response mode is `GET_N` (see below).
The response mode is one of

`GroupRequest.GET_FIRST` Returns as soon as the first response has been received (or a timeout has elapsed, or a suspicion message was received).

`GroupRequest.GET_ALL` Returns when responses from all members have been received (or a timeout has occurred, or a suspicion message was received).

`GroupRequest.GET_MAJORITY` Returns when the majority of the members have responded (or a timeout occurred, or a suspicion message was received). The majority is computed using the membership given in the constructor. If a suspicion message is received, the majority is computed anew (dynamically), excluding the suspected member.

`GroupRequest.GET_ABS_MAJORITY` Returns when the majority of the members have responded. Contrary to the above mode, the majority is *not* re-computed when a member crashes.

`GroupRequest.GET_N` Returns when n responses have been received. If n is greater than the number of members in the membership, then the call will block forever (unless a timeout is specified).

`GroupRequest.GET_NONE` Returns immediately. The message is sent asynchronously, no responses are expected, and none will be generated at the receivers' end.

Method `Execute` returns true if the group request succeeded, and false in case of an error. `Reset` allows to reuse the same group request object, by just adjusting the membership, and resending the same message (note that there are other reset methods). `GetSuspects` returns a vector of suspected members (empty if none were suspected), and `GetResults` returns a response list `RspList` which can be queried for the response from each member, whether a

member did not send a response, whether it crashed, and – if a response was received – its value.
Note that the group request relies on a loss-less message delivery: no response may ever be lost unless the sender crashed. This property must be guaranteed by the underlying transport, by for example adding a retransmission layer to the protocol stack. If this is not the case, a group request may block forever, e.g. if a response was generated by a receiver, but was lost, and the receiver did not crash (unless a timeout was specified).

3.3 MessageProtocol

A `MessageProtocol` is derived from `Protocol` and enhances it with synchronous group request functionality, using the previous two building blocks. A protocol layer in need of peer functionality would typically inherit from this protocol instead of `Protocol`.
The architecture of `MessageProtocol` is shown in fig. 3.

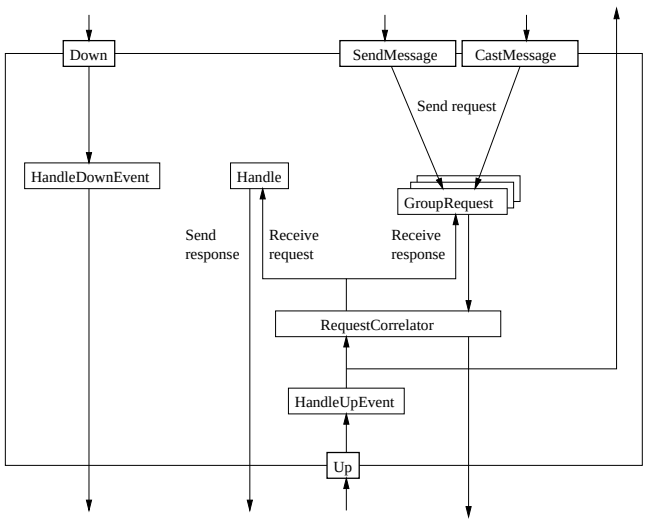

Figure 3: Architecture of `MessageProtocol`

Methods `Up` and `Down` are inherited from `Protocol`, but they are made *final*, i.e. they cannot be overridden in further subclasses of `MessageProtocol`. To handle events, methods `HandleDownEvent` and `HandleUpEvent` are introduced: `Up` will call `HandleUpEvent` and, if the return value is true, pass the message up the stack, or discard it. The same applies to `Down`: if `HandleDownEvent` return false, the message is discarded, otherwise it is passed down to the next layer.
Methods `SendMessage` and `CastMessage` provide synchronous group messages (discussed below): they make use of `GroupRequest`s and a `RequestCorrelator` (see 3.2 and 3.1) to do so. Any time one of the two methods is called, a new group request will be created and registered with the request correlator. The group request object then uses `RequestCorrelator.SendRequest` to send a message. The request correlator will assign a unique request number to the request, add a header to is so that it will be received by the correct peer layer in the receiver's protocol stack, and passes it down the stack.
The peer layer in the receiver's stack receives the message, and calls `HandleUpEvent` to do any local processing based on the message. If false is returned, the message is discarded. Else method `Receive` of the request correlator is called. It checks the header to see whether the message was generated by the corresponding peer layer in the sender's stack and, if so, receives it. Otherwise it rejects it by returning false. In this case, the message is just passed on to the layer above the current one. Otherwise, the header is scrutinized to check whether

the message is a response or request. If it is a response, the corresponding request is looked up and the message is delivered (calling method `RspCollector.ReceiveResponse`). Otherwise, method `Handle` (which is overridden by a subclass) is called. It processes the request and returns an object. If the request was not one-way, i.e. requires a response, the object is serialized and send back in a response message.
The important methods to implement a peer protocol are `CastMessage`, `SendMessage` and `Handle`:

```
public RspList CastMessage(Vector dests, Message msg, int mode, long timeout);
public Object SendMessage(Message msg, int mode, long timeout)
             throws Timeout, Suspected;
public Object Handle(Message req);
```

`CastMessage` sends message `msg` to all members of `dests` and waits for `mode` responses or a timeout (whichever comes first) and returns a response list (cf. 3.2). Mode can be any of the following:

FIRST Wait only for the first response and discard all others.

ALL Wait for all responses.

NONE Wait for no response. This makes the message one-way, i.e. no response will be generated at the receiver.

MAJORITY Wait for the majority of all receivers to respond.

When waiting for any responses, and a group member has been suspected, then that response is marked as suspected in the response list, without causing the caller to block indefinitely. Alternatively, a timeout can be employed: when `mode` responses are expected and not all have been received yet, when the timeout occurs, the call returns and the resulting response list contains all the responses received so far. A timeout of `0` means to wait indefinitely.
`SendMessage` is similar to `CastMessage` except that the message is sent only to a single group member. Argument `mode` should be either `NONE` or `FIRST` (although `ALL` and `MAJORITY` would also work). Instead of returning a response list, an object is returned, representing the response from a single receiver (may also be null). If the receiver is suspected, or a timeout has occurred, then the corresponding exception is thrown.
Callback `Handle` will probably be overridden by a peer protocol that processes incoming requests. It accepts the request in the form of a message. The body of the message will contain the request data, e.g. a request object. Depending on the request, `Handle` will perform some processing and optionally return a result value (or null). If the request was not one-way, a response message will be generated (with the result in the message's buffer) and sent back to the sender of the request.

3.4 RpcProtocol

This class is similar to `MessageProtocol`, but instead of sending and receiving *messages*, it invokes remote *methods* in its peer protocols. It is derived from `MessageProtocol` and makes use of the above mentioned methods.
Fig. 4 shows how the `RpcProtocol` class works conceptually.
In the example, the group of peer protocols would consist of the 3 members P, Q and R. Each is a protocol layer in a different protocol stack (e.g. on a different machine). P now wants to call method `A` of all members in the group and collect the responses. It does so by invoking one of the `CallRemoteMethods` methods of `RpcProtocol` (discussed below). A `MethodCall` object (see 4) will be constructed, inserted in the buffer of a message and sent to all peers.

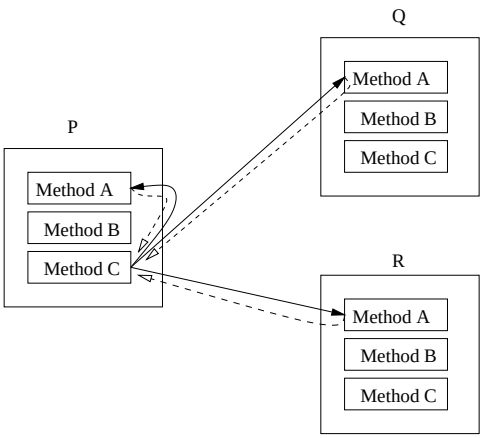

Figure 4: Remote method invocations between peer protocols

Each peer receives the message, extracts the method call object, invokes it against itself and returns a result in the form of an object (or null, if the method has a void return value). The results will be received by the calling method as a response list.
Note that, since the sender is invoking methods in all members of the group, *its own* `A` *method will also be invoked*. Therefore special care has to be taken to avoid deadlocks (see 5).
A typical protocol based on `RpcProtocol` would define a number of methods and then use one of the `CallRemoteMethod(s)` methods to invoke them. The `CallRemoteMethod` series invokes a method in a single receiver, while the `CallRemoteMethods` does so in all group members. The former takes a destination address, the method name, a number of arguments (may be zero), a mode and a timeout. The last 2 parameters are the same as for `MessageProtocol`. If the number of parameters is too large, a `MethodCall` object may be created directly and given as parameter.
The implementation of `CallRemoteMethods` is as follows:

```
public RspList CallRemoteMethods(Vector dests, MethodCall call,
                                 int mode, long timeout) {
    byte[]   buf=null;
    Message  msg=null;

    try {buf=Util.ObjectToByteBuffer(method_call);}
    catch(Exception e) {
        return null;
    }
    msg=new Message(null, null, buf);
    return CastMessage(dests, msg, mode, timeout);
}
```

The method name and arguments of the other `CallRemoteMethods` methods are used to create a `MethodCall` object. It is then serialized into a byte buffer which is inserted into a message. `MessageProtocol.CastMessage` is then used to send the message to all destinations and receive the responses.
Method `Handle` of `MessageProtocol` is overridden in `RpcProtocol` to invoke a method in the current peer:

```
public Object Handle(Message req) {
    Object      body=null, retval=null;
    MethodCall  method_call;
```

```
    if(req == null || req.GetBuffer() == null)
        return null;

    try {body=Util.ObjectFromByteBuffer(req.GetBuffer());}
    catch(Exception e) {
        return null;
    }

    if(body == null || !(body instanceof MethodCall))
         return null;

    method_call=(MethodCall)body;
    retval=method_call.Invoke(this, method_lookup_clos);
    return retval;
}
```

First, the message's byte buffer is extracted and converted into a `MethodCall` object. If this fails, `null` will be returned as result[3]. Then the method call is invoked against the current `RpcProtocol` instance (the one that received the method call) and the result returned.

4 Dynamic Method Invocation

The `RpcProtocol` class uses a `MethodCall` object to invoke methods in remote objects. The latter essentially carries a method name and an argument list, and provides operations to invoke itself against a target object, looking up the correct method to be invoked using the method name and argument list in conjunction with Java's Core Reflection API [Sun96].
When a peer wants to invoke a remote method, it calls one of the `CallRemoteMethod`(s) defined in its parent class `RpcProtocol`. They create a `MethodCall` object, embed it in a message and send the message to its destination (one or many receivers).
The receiver will extract the `MethodCall` object and look up the corresponding method: this is done using the method's name and the number and types of arguments. The method lookup method can be customized, it is for example possible to lookup methods based on the type of the receiver object *and the types of all arguments* using the `MethodLookupClos` class (similar to CLOS [Ste90]).
Since method lookup and type checking of parameters is done at run-time, and not checked at compile-time, a method call may fail. In this case, an exception will be thrown that can be handled by the caller.
There are a number of constraints on dynamic method calls: first, all arguments have to be `Objects` and implement interface `Serializable` (this allows them to be marshaled into a message and shipped to the receiver). Second, compared to strongly typed method calls, `null` arguments are not allowed because there is no class in Java that represents `null`. Third, all methods to be called have to be public (same as in the strongly typed case). Fourth, if a formal parameter is of a primitive type (e.g. `int`, `float`, `char` etc), the corresponding argument will be unwrapped at run-time before comparing it to the parameter. Finally, since there doesn't exist any mechanism for *run-time casting* (e.g. from a `Long` argument to an `int` primitive type as formal parameter), *narrowing* of values is not allowed, e.g. unwrapping the `Long` argument and narrow the resulting `long` value to the formal parameter `int` is not possible[4]. The reverse case does not require any casts, therefore it is allowed (e.g. assigning

[3] `RpcProtocol` assumes that all communication is done via `MethodCalls`, therefore it will not process other objects. However, the implementation of `Handle` could be overridden to allow this.

[4] As done in `MethodLookupCLOS`. Other method lookup implementations might of course use different algorithms.

a `String` argument to an `Object` formal parameter).
Compared to RMI [RMI96], dynamic RMCs don't require a stub/skeleton compiler. The disadvantage is that – as there is no strong typing – method calls may fail at run-time (e.g. method is not found). However, generating strongly typed client stubs from a class that uses dynamic RMCs should be easy: a compiler would parse the class for which a stub is desired (using the Core Reflection API) and generate a strongly typed method for each dynamic method. When a strongly typed method is called, its implementation would dispatch a dynamic RMC. By the same token, RMI's implementation could be extended, using dynamic RMCs to dispatch method calls not just to a single, but to multiple servers.

5 Deadlock Detection

Synchronous group messages / method calls are a powerful tool: they raise the peer protocol abstraction level to that of normal *method invocations*. Protocols can therefore be described as a set of simple procedures (methods); these procedures can be invoked remotely, and procedure bodies contain the protocol logic. Not having to write complex code for low-level tasks such as request/response correlation makes the resulting code more clear and shorter. Looking at a protocol layer as a class with methods makes understanding it much easier. Thus, an interaction between two peer protocols boils down to a remote method invocation. However, as with all types of concurrent and distributed processing, there is the danger of deadlocks. Aggravating the problem is the requirement of ordering in group communication settings: a method invocation caused by the reception of a message may not be 'passed' by another method invocation. Therefore, unlike in modern object request brokers (ORBs), where every request may be served on a separate thread (taken from a thread pool) which avoids deadlock in most cases, we cannot introduce concurrency to handle requests, since this would adversely affect ordering guaranteed by the protocol stack[5].
To illustrate the problem, consider a possible protocol interaction performed in a GMS layer when P wants to join a group (consisting of Q and R), as shown in fig. 5.

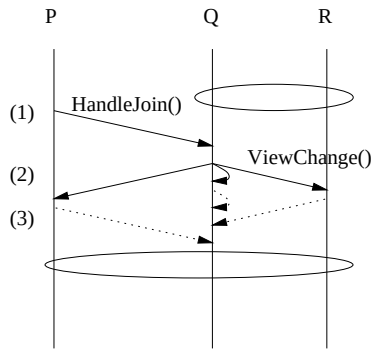

Figure 5: Client joining a group

To join the group, P invokes a unicast *asynchronous* `HandleJoin` in the coordinator (Q). The method returns immediately because it is non-blocking, and P waits until is receives a view change in which it is a member. This means that P has been added to the group successfully. Otherwise, P times out and retries its `HandleJoin` request until successful, or until there are no more members in the existing group.
When a `HandleJoin` method is received by the coordinator (1), it adds the new member to its local view and invokes a *synchronous* `ViewChange` method in all members (2). When the

[5]Note that in some cases ordering may not be strict (e.g. no FIFO ordering required). In such a case, concurrency could be introduced.

`ViewChange` methods return (3) from all members, then the join has been successful.
However, choosing to invoke the `ViewChange` method *synchronously* in all members leads to disaster in our protocol stack, which delivers the messages (hence the method invocations) in FIFO order: when the `HandleJoin` method is invoked in the coordinator, and the coordinator multicasts a synchronous `ViewChange` method invocation to all members (including itself !), the coordinator is blocked waiting on the `ViewChange` method invocation *to itself* to return (which waits to be processed). This is shown in fig. 6.

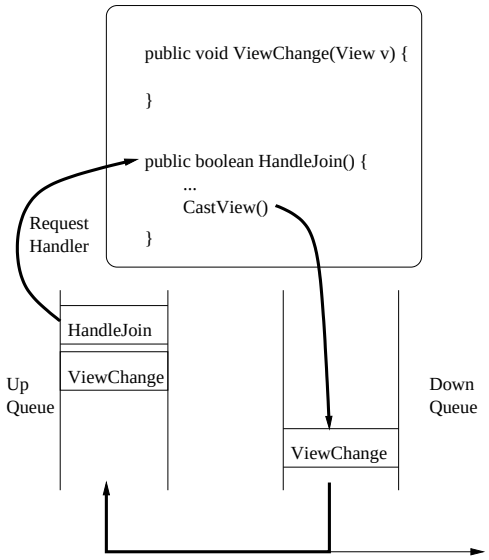

Figure 6: Request handler queue in a deadlock situation

As the GMS protocol is based on `RpcProtocol`, there are two queues: one for storing incoming methods and one for sending outgoing methods. When a request (method invocation) is received it is added to the up queue. A request handler thread continually retrieves methods from the up queue and invokes them *synchronously*, waiting for completion before handling the next method.
In the example, the queues at the coordinator are shown. A client sent an asynchronous `HandleJoin` method which, as it is now at the top of the queue, is removed from the queue and processed. The `HandleJoin` method at some point invokes a synchronous `ViewChange` on all members. To do so, a message containing the `ViewChange` method call is put in the down queue to be sent down the stack. Since the view is sent to all members, it will also be received by the *sender* (that is the coordinator) itself : it is added to the up queue, waiting for the request handler thread to remove and process it. However, the request handler is still busy processing `HandleJoin`, waiting for it to return[6]. `HandleJoin` in turn is waiting for the `ViewChange` methods it sent to all members to return: only after all the `ViewChange` methods have returned will `HandleJoin` return.
This is a clear deadlock situation, caused by the synchronous `ViewChange` method invocations. The thick lines in the example show the recursion causing the deadlock.
Note that this problem might not occur in multithreaded servers: since the `ViewChange` method would be executed on a different thread, `HandleJoin` would receive all responses and thus not block. However, as our protocol stack has to observe ordering, we can only allocate a single thread per request, ensuring FIFO order of request processing.

[6]Note that although `HandleJoin` is invoked by the client in a non-blocking manner (that means the callee does not have to return a result), the callee still has to process the request to completion before processing the next request.

As can be seen, one has to be very careful when constructing method call chains containing synchronous method calls. A certain degree of recursion is always involved in distributed group communication systems, as requests sent to the group will always also be received by the sender (unless local delivery is turned off).
As we can see, most deadlock problems occur when a message in the up queue blocks the request handler thread from processing other requests. The down queue is in most cases never a problem, as requests are just passed down the stack and the put on the network.
There are a number of workarounds to the above problems. First, careful design of protocols may reduce the number of synchronous method calls by simplifying the interactions between peer protocols. In the above example, the `ViewChange` method invocation could be made asynchronous, i.e. the `CastView` method would return immediately after sending out the `ViewChange` method invocations, without waiting for responses. In this case, no deadlock would occur. The purpose of having made `ViewChange` synchronous, namely to receive confirmation from each member that it *received* the view, can be implemented by having an acknowledgment/retransmission layer somewhere below the GMS layer: messages sent will always be received by all (non-faulty) members.
A second solution would be to have a separate thread invoke the `ViewChange` on the group: `CastView` would spawn a new thread (or reuse a thread from a thread pool) to handle the view installation and return after the thread has been created.
A third solution to the above problem is to enable the `RequestCorrelator` to detect deadlocks by setting a flag. This is described below.
The `RequestCorrelator` class allows to give higher preference to *circular* requests that would cause deadlock, thus breaking potential deadlock situations. Circular requests are chains of synchronous requests where a sender occurs more than once in the chain, e.g. a synchronous request sent from P to Q which, upon reception, sends a synchronous request to R, which in turn sends a synchronous request to P would form the call chain $P \rightarrow Q \rightarrow R \rightarrow P$. When P receives the request, it cannot be processed because P *sent* the request and is waiting for Q's response, which is waiting for R's response, which in turn is waiting for P's response. The circularity causes a deadlock. This is shown in fig. 7[7].

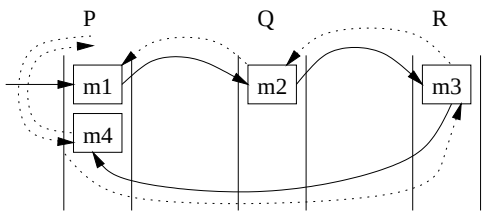

Figure 7: Circular deadlock situation

Caused by the reception of another message (or an event) m1, P sends m2 to Q, which sends m3 to R, which sends m4 to P, causing a deadlock. However, we can see that, if message m1 was preempted by m4, that is if m1 was suspended temporarily and m4 was processed instead, m4 would return (shown by dotted lines), which would cause m3, m2 and m1 to return in turn, thus avoiding deadlock. There are two problems involved with this scheme: first, we have to detect when to *prioritize* a request and second, we have to make sure that ordering guarantees given by the stack are not violated by this prioritizing scheme.
The first problem is solved by tagging synchronous requests with their *call chain* (also called *call stack*): when a synchronous request is sent, the sender's address is pushed onto a *call chain stack* in the header generated by the `RequestCorrelator`, and the header added to the message. In the example above, when m1 is received by Q, the call chain would be `P`[8].

[7] A similar (circular) deadlock situation is shown in fig. 6.
[8] For simplicity excluding the sender of the request that caused P to send m1.

Q would then add its own address to the call chain of m2, so that m2's call chain would be P → Q and so on. When P finally receives m4, m4's call chain would be P → Q → R → P. On every reception of a synchronous request, the receiver checks *whether its own address is already contained in the call chain.* If this is the case, the request is prioritized, that is, added to the head of the request queuee, otherwise it is just added to the tail. Thus, when P receives m4, m1 will be suspended, m4 processed and a response returned. Then the processing of m1 will resume[9]. This scheme allows to detect direct or indirect circularity in synchronous call chains, and to avoid deadlock by selective prioritization of requests[10].
The second problem requires examination of ordering properties. Ordering properties, such as FIFO or total order, guarantee that, if m1 was sent before m4, then m1 should be delivered before m4. However, in our case, there are two solutions: the definition of *delivery* which, in its strongest form, requires a request to *return* to complete delivery, could be weakened to one which includes delivering a request, but excludes returning from the request. The choice of definition is up to the application, some applications need strong ordering, whereas for others the weaker definition is sufficient. A stronger argument can be made when considering call chains: a call chain maintains *causality* for the requests contained in it. Since P sent m1, which indirectly caused m4, m4 can without harm interrupt m1 at P and return before m1. This can be compared to a purely local execution: processing of m1 would start before processing of m4, but m4 would 'pass' m1 and return first, causing the stack to unwind. Note that causality maintained in call chains is not a *potential* one[11], but a *real* one: a call chain is only created when – during the processing of a synchronous request – another synchronous request is sent to a different member, therefore directly relating the previous request to the next one.
The `RequestCorrelator` can be created with or without deadlock detection (default is without): there is a constructor that allows to set deadlock detection, or method `SetDeadlockDetection` can be used alternatively. When deadlock detection is off, no call chains will be added to synchronous requests, therefore neither procesing cycles nor memory is wasted. When it is on, only *synchronous* requests will be tagged with call chains by the `RequestCorrelator`, and prioritization may be used for requests causing potential deadlocks.

6 Implementing a Group Membership Protocol using Remote Method Calls

This section will show how the protocol interaction presented in fig. 6 is implemented in the group membership (GMS) layer of JavaGroups. The interaction involves *roles*, which are defined as different code (implementation) a member runs, depending on its role. Roles are *client* (a process wishing to join a group, but not yet a member), *coordinator* (existing member in charge of joining new members and sending view changes to all members) and *participant* (existing member, but not coordinator). When a process switches its role, it will also switch the code needed to perform the role. The *State* pattern [GHJV95] is used to represent roles.
When a client wants to join a group, `Channel.Connect()` is called, which in turn generates a `CONNECT` event down the protocol stack. The GMS layer catches and handles the event by interacting with other GMS layers (specifically the one that is currently coordinator). The code below shows the code for a client that is executed when a `CONNECT` event is received (no

[9] This is done using a special *scheduler* inside the `RequestCorrelator`, which accepts priority requests by suspending the current request and processing the priority request. Priority requests may preempt other priority requests recursively, until the thread pool is exhausted.

[10] Note that only *synchronous* requests are tagged with call chains, asynchronous requests will never cause the sorts of deadlocks described above, because the caller does not wait for their completion.

[11] This is different from *causal order* [Bir96, ch. 13.12.6] which records potential causality between message reception and sending, as it has no means to detect real causality.

error handling is shown):

```
public void Join(Object mbr) {
    Object    coord=null;

    while(!joined) {
        FindInitialMembers();
        if(initial_mbrs.size() < 1) {
            gms.view_id=new ViewId(mbr);  // create singleton view
            gms.members.Add(mbr);
            gms.PassDown(new Event(Event.VIEW_CHANGE,
                         gms.MakeView(gms.members.GetMembers(), gms.view_id)));
            gms.SetImpl(RpcCoordGmsImpl.CreateInstance(gms));   // become coordinator
            gms.PassUp(new Event(Event.BECOME_SERVER));
            gms.PassDown(new Event(Event.BECOME_SERVER));
            break;
        }

        coord=FindCoord(initial_mbrs);

        synchronized(view_installation_mutex) {
            try {
                gms.CallRemoteMethod(coord, "HandleJoin", mbr,
                                     GroupRequest.GET_NONE, 0);
                view_installation_mutex.wait(gms.join_timeout);
            }
            catch(Exception e) {continue;}

            if(joined)  return;     // --> success
            else        continue;   // --> failure, retry as long as there are members
        }
    }
}
```

The client first obtains an initial list of group members (e.g. by broadcasting to a well-known IP multicast address). Each response contains the address of the responder and the address of (what the responder thinks is) the current coordinator. If no response is received, then a singleton group (consisting of only the joiner) will be created and a view change propagated up and down the stack. Then, the role is switched to coordinator and another event sent up/down the stack to signal to interested layers that the process is now a fully functioning group member.
If there are other existing members in the group, the coordinator is determined and sent a unicast, asynchronous `HandleJoin` method. This is done by calling `CallRemoteMethod` (inherited from `RpcProtocol`) with the coordinator as target address. The other arguments are the name of the method to be invoked (`"HandleJoin"`), optional arguments (`mbr`), the mode indication (`GET_NONE` $\rightarrow$ asynchronous) and a timeout (0 $\rightarrow$ wait indefinitely). The caller waits for `join_timeout` milliseconds or until the `view_installation_mutex` is notified, whichever comes first. The latter will be notified when a view is received of which the caller is a member (see `HandleViewChange` below). If the view is received, the join was successful and the client code returns. Otherwise, it continues looping.
The `HandleJoin` method of the coordinator is shown below (simplified, no FLUSH processing shown):

```
public synchronized boolean HandleJoin(Address mbr) {
    View      v;
    ViewId    tmp_id;
```

```
    if(gms.members.Contains(mbr))
        return true;  // already joined

    gms.members.addElement(mbr);
    v=gms.GetNextView(gms.members, null);    // compute new view
    tmp_id=new ViewId((Address)v.GetCreator(), v.GetId());

    CallRemoteMethods(v.GetMembers(), "HandleViewChange",
                      tmp_id, v.GetMembers(), GroupRequest.GET_ALL, 0);

    return true;
}
```

It first checks whether the new member is already joined. If this is the case, the method returns. Otherwise, the new member is added to the local view. Then a new view id (address of view issuer plus logical time) is computed and sent to all members (including the coordinator), by invoking remote method `HandleViewChange` in all members. This is done using `CallRemoteMethods`, with the group membership as target destinations and the new view as argument.
When the coordinator receives its own multicast group method, its `HandleViewChange` method will just install the new view (same as for participants).
The joining client, however, reacts differently:

```
public void HandleViewChange(ViewId new_view, Vector mems) {
    if(gms.local_addr != null && mems != null && mems.contains(gms.local_addr)) {
        synchronized(view_installation_mutex) {  // wait until JOIN is sent (above)
            joined=true;
            view_installation_mutex.notify();
        }
        gms.InstallView(new_view, mems);
        gms.SetImpl(RpcParticipantGmsImpl.CreateInstance(gms));  // become participant
        gms.PassUp(new Event(Event.BECOME_SERVER));
        gms.PassDown(new Event(Event.BECOME_SERVER));
    }
}
```

If the new view includes the joiner, then `joined` is set to true and the `view_installation_mutex` is notified, causing the client's `Join` method to return successfully. Then the new view is installed and the role changed to the one of a participant. Finally, an event is sent up/down the stack, notifying all interested layers that the client has now become a fully functioning group member.

7 Conclusion

Dynamic remote method calls have a number of benefits for protocol development: they allow to structure peer protocol interaction similar to interaction between classes in object-oriented programming. A set of public methods defines the interface provided by a peer layer (server role) and a peer may itself at any time call methods of (one or more) remote peers (client role). Moving from `Protocol` which provides asynchronous messaging to `MessageProtocol` offering synchronous message sending/response reception on to `RpcProtocol` which enhances the previous with remote method invocation, means decreasing the code size of a protocol layer and increasing the abstraction level. Along with reduced code size come better understanding of a protocol, better reasoning about its correctness and improved maintainability.

The high abstraction level of RMCs is beneficial for protocol development, however, in some cases synchronous execution collides with FIFO-like ordering properties enforced by a group communication protocol stack. With the exception of very simple peer interaction schemes, synchronous RMCs have a high probability of running into deadlocks. To eliminate this problem, we presented a simple deadlock detection mechanism that tags synchronous RMCs and preempts currently processed method invocations if needed to avoid deadlocks caused by circular call chains. The caveat, however, is that this mechanism cannot detect all kinds of deadlocks. For example, similar to local concurrent programming, it is possible that two processes are competing for the same resource (protected by a lock): preempting one of the two will not help eliminate the problem. For a further discussion of deadlock situations see [Lea96].
The difference from RMI is that (a) no client stubs are used; instead *dynamic* methods calls can be assembled and invoked at run-time and (b) in addition to invoking a method in only a single server, RMCs allow to invoke group methods and collate the result(s). The lack of stubs/skeletons makes for real peer interaction with processes dynamically switching roles, whereas stubs imply a client role and skeletons a server role.

References

[Ban98] Bela Ban. *JavaGroups User's Guide.* CS Dept. Cornell University, April 1998. http://www.cs.cornell.edu/home/bba/user/index.html.

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

[CS 97] CS Dept Cornell University. *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.

[Lea96] Doug Lea. *Concurrent Programming in Java.* Javasoft, 2nd edition, 1996.

[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.

[Rit84] D. M. Ritchie. A Stream Input–Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, Oct. 1984.

[RMI96] Sun Microsystems Inc. *Java Remote Method Invocation Specification*, 1.1 edition, November 1996. Draft.

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

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

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