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 using synchronous group remote method call (G-RMC) to implement peer protocols in group communication settings. 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 G-RMC leads to less code, better understanding of a peer protocol and improved maintainability. However, ordering properties commonly required by group communication stacks may interfere with the synchronous calling of remote methods, causing deadlocks. We will present a solution that detects and resolves such deadlocks.

1 Introduction

A group is an identifier (or handle) for a number of processes that are addressed as a single entity. Processes may join and leave a group. A group can be queried for its membership and notifications are emitted when the latter changes. When a member crashes, it will eventually be removed from the group and the membership adjusted. Each member maintains a list of processes (a *view*) that it believes to be members of the group. A group membership service (one designated member) generates new views and broadcasts them to the group members when the membership changes. All non-faulty members receive the same sequence of views, and the set of messages between two consecutive views is guaranteed to be the same. This model of group communication, which we employ for this paper, is called *Virtual Synchrony* [Bir96].

Messages sent to a group (broadcast) are received by all of its members, but members may also communicate directly with single other members (unicast). Communication is implemented using a protocol stack which (among other things) ensures that all members receive the same messages sent to them in the same order. A protocol stack consists of a number of layers, each of which performs a single task.

Protocol layers often communicate with 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 provides invocation of remote methods in one or more group members and returns the result(s) synchronously (blocks until all return values have been received): *group remote method calls (G-RMC)*. They permit the developer to define protocols like regular classes known from object-oriented programming: a protocol layer is an instance of a class, containing a number of methods. It may make remote method invocations on other instances (client role), and can itself be invoked by other peers (server role). Modeling peer interaction as remote method invocation leads to a higher level of abstraction, which contributes to better understanding of a protocol, reduces the amount 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 interfere with ordering properties of the stack.
The mechanism discussed in the paper is used in the JavaGroups project [Ban98a], mainly to implement peer protocols. However, it can also be used to implement applications.
We start by briefly describing the JavaGroups toolkit, which is the group communication substrate that establishes the context for our work on synchronous G-RMC. The main section will focus on the building blocks used to implement G-RMC. Then, we will give an example of a protocol layer that uses G-RMC to communicate with its peers. Finally, we will look into the problem of deadlocks resulting from the conflicting forces of synchronous G-RMC and message ordering and offer a possible solution to prevent deadlocks while still preserving ordering properties.

2 JavaGroups Protocol Stack Architecture

JavaGroups is a group communication toolkit based on a layered protocol stack architecture similar to [Rit84, PHOA89, VRBM96, Hay98]. 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 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 is an object consisting of a receiver and a sender field, which are objects denoting the destination and source addresses. 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 a 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 a layer to attach information to a message without altering the latter.

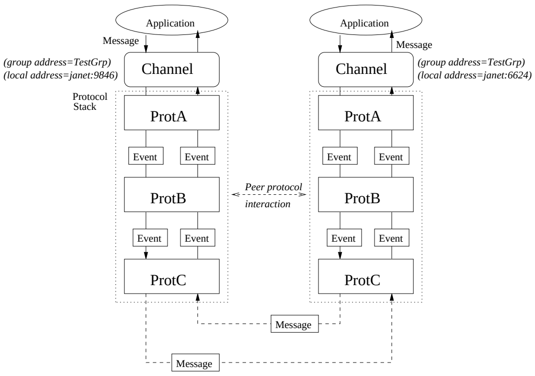

Figure 1: Architecture of JavaGroups

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 the application sends a *message* using the channel, the latter wraps it into an event and passes the event down the stack. The bottom layer extracts the message from the event and puts it on the network. When a message is received, it is again wrapped into an event and passed up the stack to the channel which extracts the message and delivers it to the application. Note that whereas *messages* are exchanged *between* different stacks, *events* are only used to transport data *within* a protocol stack. There are a number of different events such as view change, connection/disconnection and suspicion events.[1] An event never leaves the stack.

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)` and `Down(Event)`. The former is called when an event is received from the layer below, the latter when an event is received from above. The default action is to forward the event unchanged in the direction it was traveling when received.

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 respective `Up` or `Down` methods. Event processing for a queue is not concurrent: only a single event is processed at a time.

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 former type of interaction can be called *vertical interaction*, and typically consists of the following activities:

- pass a message from a layer to its neighbor (up or down the stack) unchanged,
- add a header to the message, but do 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. layer checking for duplicate messages)

These tasks can easily be done with protocol layers inheriting from `Protocol`.

The second type of protocols interact primarily with their equivalent peer instances *in different stacks*, as shown in Fig. 1 between the `ProtB` layers. Therefore this type of interaction is called *horizontal* or *peer* 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.

Note that there is a difference between *reception* and *delivery* of a message: a message is received by a layer from the layer above/below it, and is delivered to the layer below/above it when done processing. This may involve reordering the message, e.g. messages *received* as m2 and m1 may be reordered by the layer and *delivered* as m1 and m2.

For a more detailed description of the JavaGroups protocol stack refer to [Ban98a].

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, reflecting our belief 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 provides *building blocks* that hide the use of channels from the application and offer a more sophisticated interface to the application.

[1] A suspicion event is generated by a failure detector when a member is suspected of being faulty.

G-RMC is a building block built in turn from finer-grained and more general building blocks. We will 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 one 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, 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 different threads). This means that – besides sending requests and receiving the corresponding responses – every member must be able 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 required, e.g. all, majority or 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.
Essentially, a `RequestCorrelator` has one object that is responsible for handling all requests (`RequestHandler`), and zero or more *response handlers* (`RspCollector`) which will receive the response(s) for a specific request. Each request is identified by a unique ID 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 one to register a request handler which will process all requests directed towards this request correlator.
The following actions take place when a group request is sent:

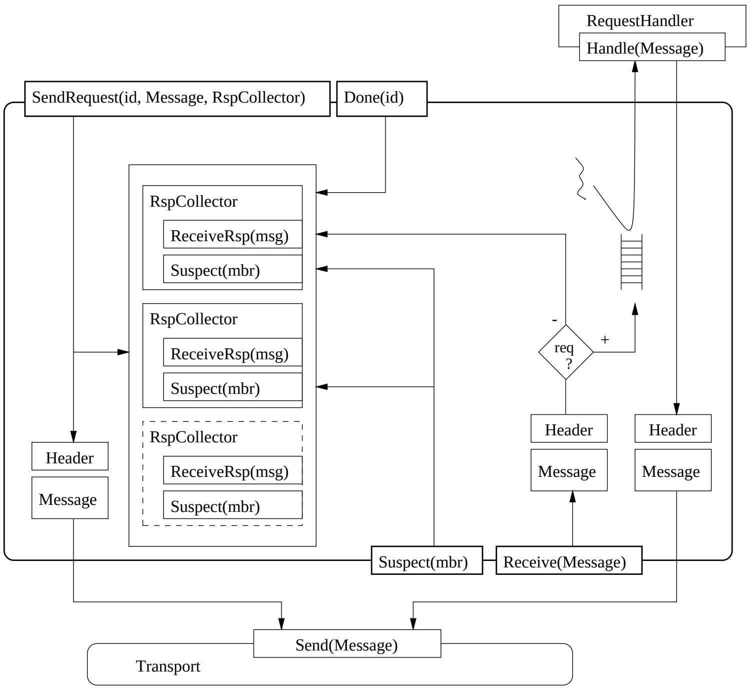

Figure 2: Request correlator

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 the latter 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 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 or not a response is expected. This is determined by `SendRequest`: if a non-null response collector is given as argument, this field will be true, otherwise it is false. The receiver of a request can therefore determine whether or not to return a response.
4. A name associated with this specific instance of request correlator. Every request correlator has a unique name to differentiate between several correlators within the same stack (see below).

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, the `Receive` method is called. It peeks at the header and checks for two 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 the next layer, e.g. the layer above the current one (when used in a protocol). The name comparison is needed to differentiate between several request correlators in different protocol layers of the same stack.

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 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 avoid blocking waiting for responses from crashed members. See section 3.2 for an example.

3.1.3 Handling a request and sending a response

As a peer process is both client and server at the same time, it can not only send requests to other peers (client role), but also has to handle requests received from other peers (server role). To handle requests, a request handler object has to be installed, which will be invoked whenever a request has been received (see below). The installation of a request handler is optional, and in some cases, peers that do not wish to process incoming requests, e.g. because they only act as clients, will not install a handler. In this case, requests from clients will simply not be answered. When sending a remote method call, an application using such client-only peers must take into account that it might not receive a response from all members and avoid blocking by for example specifying the number of responses expected, or by using a timeout (see 3.2).

When a request is received, it is added to a queue, where it is handled by the request handler thread. The latter removes requests from the queue and processes them in FIFO order, one at a time. It is only created if a request handler (`RequestHandler`) was given at the time of the request correlator's creation. Otherwise, requests received will just be discarded and no response sent[2].

The thread removes the message header and determines whether the message is asynchronous or not, i.e. whether a response needs to be sent. 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 needs to be returned to the sender of the message, then the thread creates a response message, inserts the return value object into the response's buffer, and adds another header to the message. The header would indicate that the message is 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 that 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

[2] Alternatively, if no request handler is given, a default handler could be installed, which returns a null response, or raises an exception if the provision of a request handler is required.

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 (e.g. by the GroupRequest, see below) 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, in some cases, 'all' could mean a majority, while in other cases it could mean 'responses from all members'. After Done has been called, subsequent responses for the same request are discarded. This is for example the case when a request is sent to all members, but only a simple majority of the responses is required.

3.2 GroupRequest

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

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

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

The constructors' arguments are a request correlator, the message to be sent, the 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. If a null membership vector is given, this means to wait for *all* responses.
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 of 0 means to wait indefinitely. In case a suspicion message was missed (e.g. when a group request is sent just after a suspicion message has been received, but *before* a view change has been seen) the group request object is notified of the view change, which allows it to adjust the number of pending responses for which it is waiting. This avoids blocking 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 when 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 non-faulty members have been received (or a timeout has occurred. The number of responses is recomputed whenever a suspicion or view change message is 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 and the call may therefore block.

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 receiver's end.

Method `Execute` triggers the group request and returns true on success and false on failure. `Reset` permits to reuse the same group request object, by adjusting the membership, and resending the same message. `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.

3.3 MessageProtocol

This and the next section describe two protocols derived from the `Protocol` class, which provide functionality to construct peer protocols. A protocol layer requiring peer functionality would typically inherit from one of these two protocols instead of `Protocol`.
Class `MessageProtocol` is a subclass of `Protocol` and enhances it with synchronous group request functionality, using the `RequestCorrelator` and `GroupRequest` building blocks. The architecture of `MessageProtocol` is shown in Fig. 3.
Methods `SendMessage` and `CastMessage` (discussed below) provide synchronous group messages: they make use of `GroupRequests` and a `RequestCorrelator` 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 pass it down the stack.
The peer layer in the receiver's stack receives the message and calls method `Receive` of the request correlator. The latter 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. If accepted by the request correlator, the header is scrutinized to check whether the

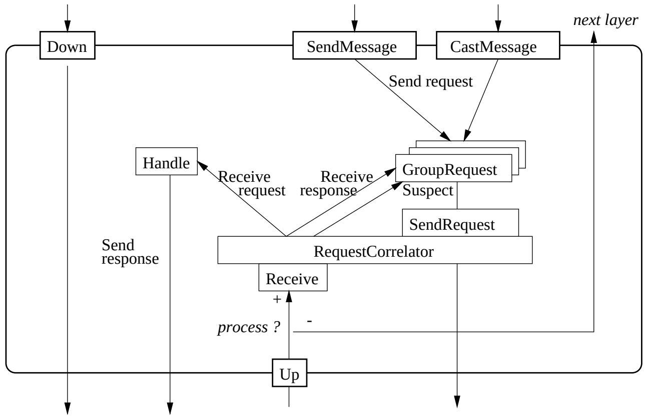

Figure 3: Architecture of `MessageProtocol`

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 asynchronous (one-way), i.e. requires a response, the object is serialized and send back to the caller 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). The message will be sent to all group members if `dests` is `null`.
`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`. 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 is the

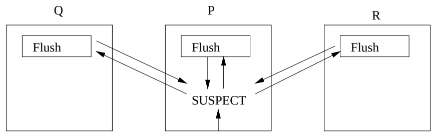

Figure 4: Remote method invocations between peer protocols

current coordinator and wants to call method `Flush` of all members in the group and collect the responses.
This could be triggered for example by receiving a `SUSPECT` event which starts a flush protocol. A flush coordinator sends a flush message to all members and receives their unstable messages as result. This interaction can be modeled as a synchronous remote method call.
The coordinator invokes 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. Each peer receives the message, extracts the `MethodCall` 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* `Flush` *method will also be invoked*. Therefore special care has to be taken to avoid deadlocks (see section 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 dsts, MethodCall c, int mode, long timeout) {
    byte[]   buf=null;
    Message  msg=null;

    try {buf=Util.ObjectToByteBuffer(c);}
    catch(Exception e) {
        return null;
    }
    msg=new Message(null, null, buf);
    return CastMessage(dsts, 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, Ban98b]).

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 `Object`s and implement interface `Serializable` (or `Externalizable`) (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 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

[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.

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 Group RMCs and Deadlocks

Synchronous group 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 readable and shorter. Looking at a protocol layer as a class containing methods makes understanding it much easier. Thus, an interaction between two peer protocols boils down to a remote method invocation.

As shown in Fig. 1, messages are passed between protocol layers in FIFO order to preserve the ordering guarantees established by the different layers. It is easy to see that the non-determinism introduced by concurrent processing of messages could violate ordering properties. Consider a layer that orders messages according to sequence numbers and passes them to the next layer in increasing order of sequence numbers: if the next layer did not receive those messages in the order the previous layer established, but in any random order, then the ordering properties of the previous layer are violated. This is the reason why FIFO queues are used to pass messages between protocol layers, and only a single message at a time is processed by a layer (in each direction).

Using synchronous G-RMCs in the context of such an ordered FIFO processing of messages may cause a problem: since the caller of a remote method is invoked *itself*, deadlock may ensue as shown in the following example. 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 (simplified protocol).

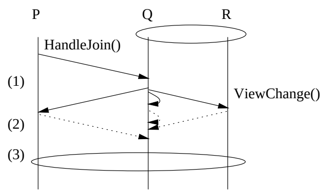

Figure 5: Client joining a group

To join the group, P invokes a remote unicast *asynchronous* `HandleJoin` in the coordinator (Q). The method returns immediately because it is non-blocking, and P waits until it receives a view change in which it is a member, meaning 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 in which case it forms a singleton 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 `ViewChange` methods return (3) from all members, then the join has been successful.

In contrast, 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.

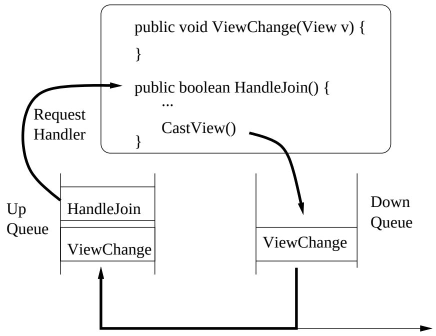

Figure 6: Request handler queue in a deadlock situation

Since 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[5]. `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 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 therefore 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.
As we can see, 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).
Most deadlock problems occur when a message in the up queue blocks the request handler thread from processing other requests. The down queue is rarely a problem, because requests are just passed down the stack and then put on the network.
There are a number of work-arounds to the above problems. First, careful design of protocols could reduce the number of synchronous method calls needed by simplifying the interactions between peers. 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. However, since the responses from the synchronous `ViewChange` RMC serve as confirmation that all members received the view, we would have to come up with a different way of ensuring that each member received the new view (e.g. an explicit asynchronous

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

acknowledgement from each receiver). As can be seen, the protocol complexity rises immediately when not using a synchronous RMC.
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, which is described below.

5.1 Deadlock Resolution

The `RequestCorrelator` class can be enabled to assign higher priority to *circular* requests that would cause deadlock, thus breaking potential deadlock situations of the kind discussed here. 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[6].

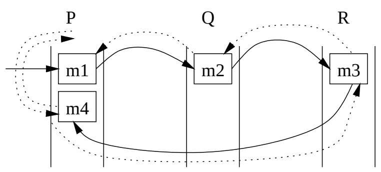

Figure 7: Circular deadlock situation

Upon the reception of another message 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 terminate successfully, 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 prioritizing in this manner.

5.1.1 Detecting Circularity

The first problem is solved by tagging synchronous requests with their *call chain* (or *call stack*): when a synchronous request is sent, the sender's address is pushed onto a call stack in the header generated by the `RequestCorrelator`, and the header is added to the message. In the example above, when m1 is received by Q, the call chain would be `P`[7]. Q would then add its own address to the call chain of m2, so that m2's call chain would be P $\rightarrow$ Q and so on. When P finally receives m4, m4's call chain would be P $\rightarrow$ Q $\rightarrow$ R $\rightarrow$ 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 queue, 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[8]. This scheme allows one to detect direct or indirect circularity in synchronous

[6] A similar (circular) deadlock situation is shown in Fig. 6.

[7] For simplicity excluding the sender of the request that caused P to send m1.

[8] 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.

call chains, and to avoid deadlock by selective prioritization of requests[9].

5.1.2 Preemption and Ordering Problems

Prioritization solves the circular method call deadlock problem by preempting blocking calls. However, it also weakens FIFO processing of calls because, by moving a peempting call to the top of the processing queue, calls that are between the call to be preempted and the one preempting it are 'passed', violating the order established by the lower layers. This is shown in Fig. 8c.

Assume that m1 triggers synchronous RMC m3, but before m3 is received, RMC m2 (sent by another process) is received. When m3 is received, it will preempt m1, which is correct, but it will also preempt m2, which is wrong, as m2 should be seen before m3. Therefore, we should pass up all messages between a blocked and the corresponding unblocking message. More generally: when a message is received that detects a circular call chain, all messages[10] between it and the message causing the circularity should be processed before preemption takes place. In the above example, when m3 is received, it is seen that it would prempt m1. Therefore, m1 is preempted with m2 first, and then with m3.

There are two issues associated with this scheme: first, since an unblocking message is always moved to the top of the stack, we have to convince ourselves that blocking messages are always at the top of the stack (otherwise we would have to move the unblocking message directly ahead of the blocking message) and second, we have to look at cases where the above scenario (other messages between blocking and corresponding unblocking messages) happens.

In the first case, it is easy to see that only the message currently processed can *trigger* a synchronous (potentially blocking) RMC, so that message has to be on top of the queue. Therefore any message that preempts another one can safely assume that the other message is at the head of the queue, and preempt it by moving to the head of the queue.

The second case (as shown in Fig. 8c) can happen when (1) a synchronous RMC is sent to the group, but before it is *received* an RMC from a different process is received, or (2) the sender sends one or more *asynchronous* RMCs before sending a *synchronous* RMC. In these cases, all non-blocked RMCs between the unblocking and the corresponding blocked RMC have to be processed before preemption, as shown in Fig. 8a.

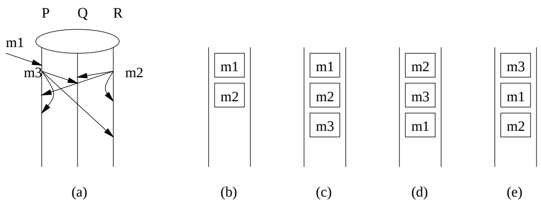

Figure 8: Messages between the unblocking and the blocked RMC

In this example, synchronous RMC m1 is received by P which causes synchronous RMC m3 to be broadcast to the group. But at the same time, R broadcasts m2. Let's assume a total ordering layer decides that m2 is to be delivered *before* m3 at all receivers. Fig. 8b shows P's up-queue after having received m1 and m2. The processing of m1 triggers synchronous RMC m3 which is received in 8c. When it is received the circularity check detects that m3 has to preempt m1 to avoid deadlock. However, m2 has to be processed *before* this preemption can take place. Therefore, processing of m1 is suspended and both m2 and m3 moved to the top of the queue (as shown in 8d). When processing of m2 is done, m3 will be processed, which

[9] 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.

[10] Only messages that are not marked as blocking are moved. A message is marked as blocking when a circularity is detected, e.g. in the example m1 is marked when m3 is received during checking the call chain.

unblocks m1. When m3 terminates, m1 is processed and terminates as well. This scheme ensures that the ordering properties of a stack, (in the above example, total ordering) are not violated by message preemption. If in-between messages were not considered, then instead of P, Q and R receiving m2 *before* m3, P would instead receive m2 *after* m3 as shown in Fig. 8e.

Taking in-between messages into account is a non-trivial task and has not yet been implemented in the current version of the toolkit. However, in-between messages are currently not a problem because the use of G-RMC in our toolkit is restricted to one member per group at a time (e.g. the coordinator in a group membership layer, which is in the client role, whereas all others act in the server role), which avoids ordering problems with RMCs sent by other members. Current experience with synchronous G-RMCs indicates that complex synchronous protocol interaction is rather rare and most situations can be handled by our current implementation.

The `RequestCorrelator` can be created with or without deadlock detection (default is without). When deadlock detection is off, no call chains will be added to synchronous requests, therefore neither processing 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. Depending on the current role, requests are delegated to different objects which all implement the same interface, but provide different implementations.

The protocol implementing GMS is derived from `RpcProtocol` since it needs to interact with its corresponding peer protocols in all members. Since the peer interaction shown in Fig. 6 would produce a deadlock, deadlock detection is turned on when the protocol is created.

When a client wants to join a group, `Channel.Connect()` is called, which in turn generates a `CONNECT` event to be sent 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 error handling is shown):

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

    while(!joined) {
        FindInitialMembers();
        if(initial_mbrs.size() < 1) {
            // Create singleton view, become coordinator (first member)
            break;
        }
        coord=FindCoord(initial_mbrs);
        synchronized(view_installation_mutex) {
            gms.CallRemoteMethod(coord, "HandleJoin", mbr, GroupRequest.GET_NONE, 0);
            view_installation_mutex.wait(gms.join_timeout); // wait for view
```

```
            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 the role is switched to coordinator.

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"`), the address of the joining member (`mbr`), the mode indication (`GET_NONE` → asynchronous) and a timeout (0 → 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 `ViewChange` 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;

    gms.members.addElement(mbr);
    v=gms.GetNextView(gms.members, null);    // compute new view
    tmp_id=v.getVid();
    CallRemoteMethods(v.GetMembers(), "ViewChange",
                      tmp_id, v.GetMembers(), GroupRequest.GET_ALL, 0);
    return true;
}
```

First, 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 `ViewChange` in all members. This is done using `CallRemoteMethods`, with the group membership as target destinations and the new view as argument.

It is important to note – that without deadlock detection – the `ViewChange` RMC would hang since the coordinator's `ViewChange` method cannot be called as it is still processing `HandleJoin`. This circular deadlock was shown in Fig. 6. However, using the simple deadlock resolution presented earlier, `ViewChange` preempts `HandleJoin` and thus progress is made.

When the coordinator receives its own multicast group method call, its `ViewChange` method will install the new view (same as for participants).

The joining client, however, behaves differently on receiving its initial view:

```
public void ViewChange(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
    }
}
```

If the new view includes the joiner, then `joined` (an instance variable) 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.

7 Conclusion

Dynamic remote method calls have a number of benefits for protocol development: they allow us 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 is prevented by 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 due to the recursion inherent in group communication. 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].

An additional but important issue is that the solution presented can only be used when the associated re-ordering of events will not violate assumptions built into the protocol stack. Here, we illustrated the idea; in general it would be necessary to undertake a careful analysis and proof on a case-by-case basis before using this approach. Nonetheless, we believe it to be powerful and broadly applicable in protocols of the sort considered in our work.

8 Acknowledgements

The author wishes to thank Ken Birman, Ohad Rodeh and Raoul Bhoedjang for comments and suggestions on this paper.

References

[Ban98a] Bela Ban. *JavaGroups User's Guide*. CS Dept. Cornell University, April 1998. http://www.cs.cornell.edu/home/bba/user/index.html.

[Ban98b] Bela Ban. Static vs. Dynamic Method Resolution in Java: The Case For Argument-Based Method Selection. http://www.cs.cornell.edu/home/bba/papers.html, 1998.

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

[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*. PhD thesis, Computer Science Department, Cornell University, January 1998.

[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.