A Simple Deadlock Resolution Scheme for Synchronous Reliable Group RPC

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

Abstract

Reliable group communication requires that messages are received by all group members in the same order. A useful abstraction for group communication is *synchronous Group RPC* which invokes a remote procedure in each member and blocks until all replies have been received. However, due to the fact that requests cannot be handled concurrently, but have to be processed in FIFO order to observe ordering, synchronous Group RPC is very likely to deadlock. A typical situation involves the receiver of a blocking RPC in turn invoking another blocking Group RPC. We describe a simple scheme for detecting potential deadlocks caused by circular synchronous RPCs and resolving them.

1 Introduction

Group communication involves aggregation of a number of processes into a logical container, called a group, that is addressed as a single logical entity. Messages sent to the group are received by all members. Most group applications require some form of ordering over the messages received, e.g. all messages sent by a certain member have to be ordered with respect to their sequence numbers (FIFO order) or all messages from all members have to be received in the same order at every receiver (TOTAL order).

A synchronous Group RPC (GRPC) invokes a procedure in all group members and blocks until all replies have been received. It is similar to regular point-to-point RPC [BN84], but extended to multi-point communication. Note that we use the traditional term of RPC, but mostly refer to *method invocation* rather than procedure invocation.

When the receiver P of a synchronous RPC m_1 in turn invokes another synchronous GRPC m_2 as part of the processing logic of m_1, then m_2 (which is also received by P as P is itself a group member) will not be processed by P until m_1 has completed (which in turn waits for the completion of m_2), causing a deadlock. Note that this only happens because we need to process each incoming request on a single thread; requests are supposed to be processed in the order received. If we instead processed each request in a separate thread, then no deadlock would occur since the processing of m_2 would be independent (i.e. concurrent) of the processing of m_1[1]. However, we cannot use concurrent request handling since the non-determinism of request processing would violate the ordering properties associated with the group.

We present a scheme for detecting and resolving deadlocks caused by scenarios such as the one described above. The mechanism was implemented and will be presented in the context of the JavaGroups [Ban98] reliable group communication toolkit, which uses a layered protocol stack similar to the xKernel [PHOA89] or Stream [Rit84] projects. However, the deadlock resolution mechanism is generic enough to be used in other group communication systems.

[1] Note that most modern Object Request Brokers make use of multi-threaded request handling.

We will start by giving an overview of the protocol stack architecture and discuss how ordering is enforced. Then we will present the GRPC abstraction and its implementation and describe a deadlock situation. (Note that the term GRPC denotes both remote group invocation and the Java class that provides this functionality). The main part will present the deadlock detection and resolution scheme.

2 Ordering in Reliable Group Communication

Reliable group communication extends reliable point-to-point communication (as e.g. provided by TCP) to multi-point settings. This means that in addition to the properties of reliable point-to-point transmission such as loss-less packet delivery, fragmentation and ordering at the receiver with respect to the sender, multi-point transmission has to ensure that messages sent to the group are seen by all receivers in the same order. The properties provided by a protocol stack are the sum of the properties provided by each protocol layer, as shown in Fig. 1.

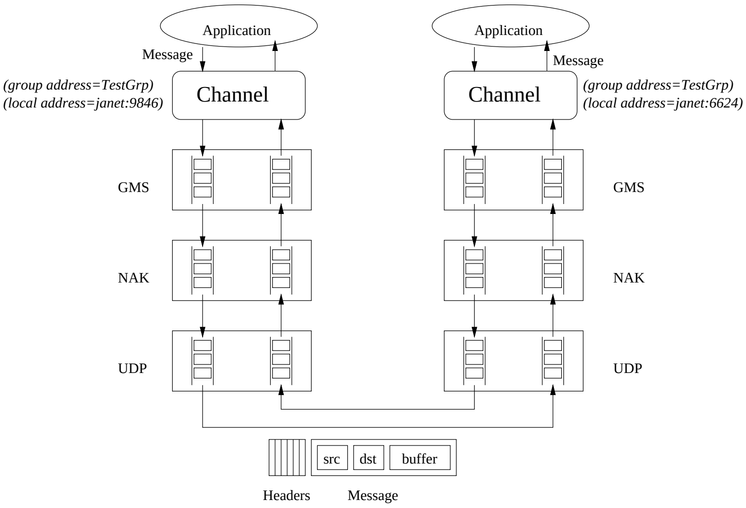

Figure 1: Architecture of JavaGroups

The composition of the stack is defined by the user when creating a channel. A channel is a group endpoint and serves as API to the application, allowing the latter to join and leave groups, broadcast messages to the group, receive messages and get notifications when the group membership changes (e.g. new members join, or existing ones leave or crash). By combining different protocols, it is possible to create protocol stacks with varying degrees of reliability, tailored to the application's requirements[2].

The protocol stack consists of a linear list of protocols which are connected by two queues: one for messages traveling from the network up to the application, and one for messages traveling down. Each protocol enforces a specific property, e.g. the NAK layer provides loss-less transmission by numbering messages and requesting retransmission of missing messages. A protocol is a subclass of a default protocol which takes care of message processing and calls the `Up()` or `Down()` method whenever a message is received. Subclasses typically override the default behavior (which just forwards the message) to implement the property provided by the specific subclass.

The default protocol processing adds a message to the tail of the corresponding up or down-queue whenever one is received from a neighbor protocol. A thread is associated with each

[2]Note that because all protocols have the same interface, it is possible to syntactically combine any protocols. However, most combinations do not make sense semantically, e.g. NAK as bottom-most protocol

queue, performing the following tasks: it peeks at the message at the head of the queue and calls the corresponding `Up()` or `Down()` method, which performs some computation on the message (e.g. it may modify, re-order, forward, discard or even trigger new messages). Protocols may also add or remove a header containing additional information relevant to the specific protocol (e.g. a checksum or public key for an encryption protocol) to/from the message. When the method returns, the message will be dequeued and the next message will be handled.

This FIFO, one-message-at-a-time processing is necessary because we do not want to violate properties imposed by the protocol layer from which we received the message. It is easy to see that a message sent from the UDP protocol to the NAK protocol may be added in any order to the NAK protocol's up-queue, as UDP does not order messages and NAK will re-order messages anyway before forwarding them up. However, forwarding a message received from NAK to GMS in any order would violate the ordering imposed by NAK, therefore FIFO processing is needed here. Since protocols are independent from each other in the sense that a protocol does not know what property its neighbor enforces, we need to process one message at a time and forward them in FIFO order. We will see in section 4 that by enforcing safety, we will not be able to guarantee liveness of GRPC, resulting in deadlocks. A solution that still guarantees safety, but additionally provides liveness by detecting certain types of deadlocks and resolving them is presented in section 4.

3 Synchronous Group RPC

Synchronous GRPC is a class in JavaGroups that allows a developer to make remote procedure calls to all members of a group and to block until n replies have been received[3]. Values for n may be all replies, the first reply only (discarding subsequent replies), and group majority. The result of the GRPC is a list of replies, with each entry containing (a) the result (if available), (b) the sender, (c) a flag stating whether the reply was received or not (e.g. when a timeout occurred, some replies may not have been received) and (d) a flag marking the reply as *suspected*. The latter means that while waiting for replies, a member from whom we hadn't received a reply yet, was suspected as having crashed, in which case we mark its reply as not received and suspected. This prevents blocking indefinitely, e.g. when waiting for replies from members P, Q and R, and R crashes. In this case, the GRPC would not have to wait for R's reply (which will never be received), but may return after having received P's and Q's reply (with R's reply being marked as suspected). JavaGroups uses a suspicion service [VRMH98] that detects when a member has crashed and emits a notification to all members. The current group coordinator will the typically exclude the faulty member from the group.

GRPC needs a transport underneath it that allows it (a) to send a message to a group or a single member, (b) to receive a message and (c) optionally to receive failure notifications. Since these are minimal requirements, GRPC can be used on top of channels, to build applications and within protocols, e.g. to build *peer protocols*. Peer protocols communicate horizontally with the same instances in different protocol stacks (e.g. a group membership service GMS), whereas regular protocols communicate vertically with their neighbors below or above. Contrary to client – server communication, where roles are static, peer communication involves switching between the client and server roles dynamically. E.g. a client invoking a GRPC (as a client) will typically receive the RPC itself since it is also a member of the group, thus has to handle it (as a server).

Here, we focus on the use of GRPC to build peer protocols. The GMS protocol (shown in Fig. 1) uses GRPC to invoke remote methods in the GMS protocols of all member stacks. The architecture of GRPC is shown in Fig. 2.

[3] Note that there is also an asynchronous, non-blocking GRPC available, which sends a GRPC and the returns immediately, without waiting for replies.

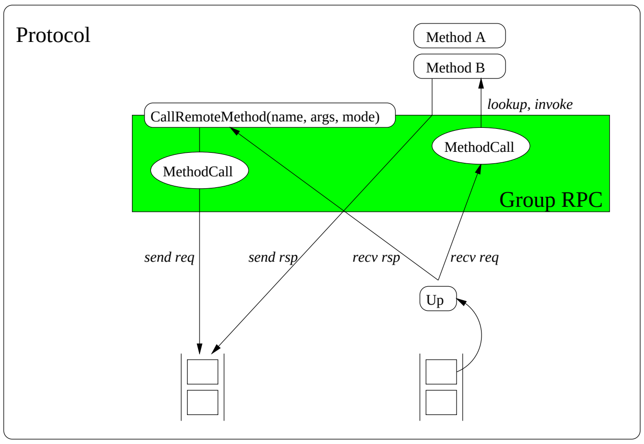

Figure 2: Group RPC

The central functionality of GRPC is to (a) invoke GRPCs and receive replies (client role) and (b) to handle RPCs received and send replies (server role). An object can register with a GRPC instance, thus being able to handle incoming RPC requests. In the example, the protocol itself is the object, so it will be invoked whenever an RPC is received, and is able to invoke remote methods using GRPC.

`CallRemoteMethod()` accepts the name of the method to be invoked, a list of arguments and a mode, which determines how many replies we have to wait for. It then creates a `MethodCall` object containing the method name and arguments, which is serialized into the byte buffer of a message. A header is added to the message, containing among other things a flag that determines whether the message is a request or reply, and a request ID which is subsequently used to correlate incoming replies with their originating request. The message is sent to all members by passing it down to the protocol below, which is done by adding the message to the protocol's down queue, where the handler thread will dequeue and forward it to the layer below.

On the receiver's side (at each member), the thread processing the up queue dequeues and handles one message at a time, calling the `Up()` method. The header of the message is removed and used to determine whether the message is a request or reply[4]. If it is a reply, it is passed back to the caller (identified by the request ID) who will eventually return after having received all (or n) replies. If it is a request, the message's byte buffer is unserialized into a `MethodCall` object. This object contains an `Invoke()` method which, given a target object[5], uses the target object's class, the method name and the number and types of its arguments to lookup the corresponding method (e.g. method B in the figure). The method is then invoked and the result returned to the invoker of the RPC by again creating a message (this time it is a response) and adding it to the down queue. Method lookup and invocation are performed using Java's Reflection API [Sun96]. The `Up()` method finally returns, allowing the thread processing the up queue to dequeue the next message. Note that – as discussed in section 2 – the handler thread must process only one message at a time, which will be the major reason for deadlocks, as illustrated below.

4 Deadlocks

Synchronous GRPC is a powerful tool: it raises the peer protocol abstraction level to that of normal *method invocations*. Protocols can therefore be described as a set of simple procedures

[4]If the message did not originate in a (peer) GMS layer, it is simply passed up.
[5]Which is the current protocol object in the example.

(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.

However, in systems where requests have to be processed single-threadedly, the possibility of running into deadlock situations is high. Two examples of such a situation are presented below.

4.1 Examples of Deadlocks

4.1.1 Group Membership Join Protocol

As discussed earlier, FIFO processing between protocols is necessary to observe the ordering established by the combination of protocols in the stack. Using synchronous GRPCs in the context of such an ordered FIFO processing may cause a problem: since the caller of a remote method is invoked *itself* (being also a group member), deadlock could 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. 3 (simplified protocol).

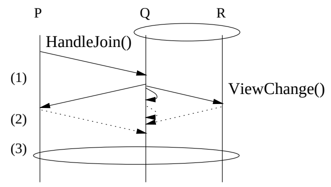

Figure 3: Client joining a group

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[6] 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 the coordinator's `HandleJoin()` method is invoked (1), it adds the new member to its view and invokes a *synchronous* `ViewChange()` method in all members in order to broadcast the new membership list to all members (2). When the `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. 4.

As previously shown, there are two queues: one for receiving messages and one for sending messages. When a request (method invocation) is received it is added to the up queue. A request handler thread continually retrieves messages from the up queue and invokes them *one at a time*, waiting for completion before handling the next message.

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 up queue, is dequeued and processed. The `HandleJoin()` method at some point invokes a synchronous `ViewChange()`

[6] A *view* is a list of members currently believed to be in the group.

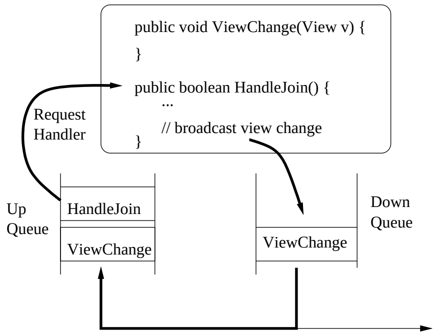

Figure 4: Request handler queue in a deadlock situation

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[7]. `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 does not occur in multi-threaded request handling: 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.

4.1.2 Distributed Callback

An example similar to the one above resulting in deadlock occurs in systems involving distributed objects (e.g. CORBA, RMI) that use single-threaded request processing. When a client invokes a remote method in a server synchronously (waiting for the result) and the server calls back the client synchronously before returning, then deadlock ensues. The reason is that the client request handler is still processing the RPC (waiting for the result to be returned), therefore cannot handle the server's callback. This problem could be solved using a thread-per-method request handling strategy, however, in certain systems, concurrent processing cannot be used, be it to save resources, or for example to reduce non-determinism introduces by concurrency, e.g. in real-time systems.

Our scheme of detecting and preventing deadlocks caused by circular requests can also be applied to solve the distributed callback problem, in cases where concurrent request processing cannot be used.

4.2 Workarounds

As we can see, one has to be very careful when constructing method call chains containing synchronous method calls. Especially in group communication systems, a certain degree of circularity is always involved, as requests sent to the group will always also be received by the sender (unless local delivery is turned off).

[7] 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.

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 Join Protocol example, the `ViewChange()` method invocation could be made asynchronous, i.e. the `HandleJoin()` method would return immediately after sending the `ViewChange()` method invocations, without waiting for responses. In this case, no deadlock would occur. However, since the responses from the synchronous `ViewChange()` RPC serve as confirmation that all members have indeed 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 acknowledgement from each receiver). As can be seen, the protocol complexity rises immediately when not using a synchronous RPC.
A second solution (to the same example) would be to have a separate thread invoke the `ViewChange()` on the group: `HandleJoin()` 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 is to try to detect circularity and resolve it in the GRPC object. The solution involves assigning higher priority to *circular* requests that would cause deadlock, thus breaking potential deadlock situations of the kind discussed here. This is discussed in the next sections.

4.3 Detecting Circular Deadlocks

Circular requests are chains of synchronous requests where a sender occurs more than once in the chain, as shown in Fig. 5.

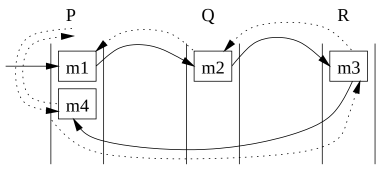

Figure 5: Deadlock situation caused by circular synchronous requests

A synchronous request m_2 sent from P to Q which, upon reception, invokes synchronous m_3 in R, which in turn invokes m_4 in P, forms 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.
We can detect circularity 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 GRPC, and the header is added to the message. In the example above, when m_1 is received by Q, the call chain would be P[8]. Q would then add its own address to the call chain of m_2, so that m_2's call chain would be P $\rightarrow$ Q and so on. When P finally receives m_4, m_4's call chain would be P $\rightarrow$ Q $\rightarrow$ R.
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, therefore they do not have to be tagged.

[8] For simplicity excluding the sender of the request that caused P to send m_1.

4.4 Resolving Circular Deadlocks

Call chains are used to solve the circular deadlock problem. The idea is to detect when the currently processed (synchronous) request m_x receives a (synchronous) RPC m_y that was (directly or indirectly) triggered by it. In this case, m_x has to be preempted by m_y; m_y must be processed before m_x, thus preventing m_x from blocking indefinitely.

This is done as follows: upon 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 m_4, m_1 will be suspended, m_4 processed and a response returned. This causes m_3 and m_2 to return successfully (shown by dotted lines). Then the processing of m_1 resumes and m_1 returns, too.

Since an unblocking message is always moved to the top of the queue, we have to convince ourselves that blocking messages are always at the top of the queue; otherwise we would have to move the unblocking message directly *ahead* of the blocking message. The argument goes as follows: blocking is local to each stack; the same RPC may block in one protocol stack, whereas it may not block in others (as shown in Fig. 4 where only the coordinator blocks). Since blocking and unblocking are local, an unblocking message can only unblock a message that triggered it. Because we are only interested in local messages that triggered the unblocking message, and triggering can only occur in the currently processed message, the triggering (therefore potentially blocking) message must be on top of the stack.

Suspension and resumption of request processing is done using a special *scheduler* inside the GRPC, 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. This scheme allows one to detect direct or indirect circularity in synchronous call chains, and to avoid deadlock by selective prioritization of requests.

Prioritization solves the circular method call deadlock problem by preempting blocking calls. However, it also weakens FIFO processing of calls because, by moving a preempting 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.

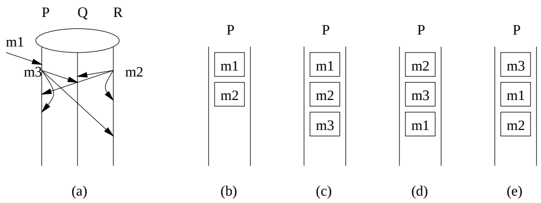

Figure 6: Messages between the unblocking and the blocked RPC

Consider Fig. 6: synchronous RPC m_1 is received by P which causes synchronous RPC m_3 to be broadcast to the group. But at about the same time, R broadcasts m_2. Let's assume a total ordering layer decides that m_2 is to be delivered *before* m_3 at all receivers. Fig. 6b shows P's up-queue after having received m_1 and m_2. The processing of m_1 triggers synchronous RPC m_3 which is received in 6c.

When it is received, the circularity check detects that m_3 has to preempt m_1 to avoid deadlock, which is correct. However, m_3 will also preempt m_2, which is wrong, as m_2 should be seen before m_3. Therefore, m_2 has to be processed *before* the preemption of m_1 by m_3 can take place.

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[9] between it and the message causing the circularity should be processed before

[9] Only messages that are not marked as blocking are moved. A message is marked as blocking when a

preemption takes place, to preserve total ordering.
Therefore, processing of m_1 is suspended and both m_2 and m_3 are moved to the top of the queue (as shown in 6d). When processing of m_2 is done, m_3 will be processed, which unblocks m_1. When m_3 terminates, m_1 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 all members receiving m_2 *before* m_3, P would instead receive m_2 *after* m_3 as shown in Fig. 6e.
Taking in-between messages into account is a non-trivial task and is not yet fully implemented in the current version of the toolkit. However, in-between messages are currently not a limitation because the use of GRPC in our toolkit is restricted to one member per group at a time, e.g. the coordinator in a group membership layer, which is the only member sending GRPCs, thus avoiding ordering problems with RPCs sent by other members concurrently. Current experience with synchronous GRPCs indicates that complex synchronous protocol interaction is rather rare and most situations can be handled by our current implementation.
The GRPC component 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 GRPC, and prioritization is used for requests causing potential deadlocks.

5 Conclusion

GRPCs 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). The high abstraction level decreases the code size of a protocol layer and increases readability of protocols. Along with reduced code size come better understanding of a protocol, better reasoning about its correctness and improved maintainability.
As GRPCs are beneficial for protocol development, 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 GRPCs have a high probability of running into deadlocks due to the circularity inherent in group communication. To eliminate this problem, we presented a simple deadlock detection mechanism that tags synchronous RPCs 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 eliminate the problem. For a further discussion of deadlock situations see [Lea96].
Here, we illustrated the idea of a simple mechanism for resolving deadlocks caused by circular group RPCs in the context of strict ordering; 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.

circularity is detected, e.g. in the example m_1 is marked when m_3 is received during checking the call chain.

6 Acknowledgements

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

References

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

[BN84] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2:39 – 59, February 1984.

[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.

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

[VRMH98] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A Gossip-Style Failure Detection Service. In Nigel Davies, Kerry Raymond, and Jochen Seitz, editors, *Middleware. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 1998.