# Constructing a Configurable Group RPC Service

Matti A. Hiltunen Richard D. Schlichting

TR 94-28

# Constructing a Configurable Group RPC Service[1]

Matti A. Hiltunen Richard D. Schlichting

TR 94-28

**Abstract**

Current Remote Procedure Call (RPC) services implement a variety of semantics, with many of the differences related to how communication and server failures are handled. The list increases even more when considering group RPC, a variant of RPC often used for fault-tolerance where an invocation is sent to a group of servers rather than one. This paper presents an approach to constructing group RPC in which a single configurable system is used to build different variants of the service. The approach is based on implementing each property as a separate software module called a micro-protocol, and then configuring the micro-protocols needed to implement the desired service together using a software framework based on the *x*-kernel. The properties of point-to-point and group RPC are identified and classified, and the general execution model described. An example consisting of detailed pseudocode for a modular implementation of a group RPC service is given to illustrate the approach. Dependency issues that restrict configurability are also addressed.

October 18, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grant N00014-91-J-1015.

# 1 Introduction

*Remote Procedure Call* (RPC) [Nel81, BN84] is a communication abstraction designed to simplify the writing of distributed programs. With RPC, a request for service from a *client* to a *server* process is structured to give synchronization semantics at the client similar to normal procedure call. Numerous examples of different RPC services and implementations exist, including Firefly RPC [SB90], Alphorn [AGH+91], lightweight RPC [BALL90], Peregrine [JZ93], and SUPRA-RPC [Sto94]. Among the commercial RPC packages released have been Courier from Xerox [Xer81], Sun RPC [Sun88], Netwise RPC from Novell Netware, NCA from Apollo [Apo89], and DCE RPC.

On the surface, the semantics of RPC seem very simple, yet the reality is that there are subtleties and variations. For example, there are many ways to define how an RPC service deals with server and communication failures. The set of options grows even more when considering *group RPC*, a variant of RPC often used for fault-tolerance where the request is sent to a group of servers rather than one. For example, there are numerous ways to define how requests are ordered at a server, and how the multiple replies to a given request are collated for return to the client. Making choices in each of these cases gives a different variant of RPC. This explains at least in part why so many RPC systems have been defined and implemented: since each system typically realizes one and only one set of semantics, a new system is built whenever different semantics are called for by the application requirements.

In this paper, we present an alternative approach in which a single, configurable system is used to construct different variants of RPC. In our approach, a user begins by analyzing the requirements of the application with respect to the different types of properties realized by RPC. They then select the desired variant of each type. For example, they might select *at least once* semantics to deal with failures [Spe82] and *orphan termination* to deal with orphaned computations [Shr83]. Finally, software modules called *micro-protocols* [HS93, BS94], each of which implements a single selected property, are configured together using a software *framework* based on the *x*-kernel [HP91]. The approach assumes an asynchronous distributed system, where the underlying communication system can experience both omission and performance failures, and where sites can experience crash failures.

Many of the properties for which multiple variants exist are related to failure handling and fault-tolerance, so we deal exclusively here with these aspects of RPC. Other issues, although important, are beyond the scope of this paper. These include stub generation and heterogeneity [Sun88, Gib87, HS87, Apo89, TB90, WSG91], binding [BN84, LT91, BALL90], performance or performance optimizations, [PA88, RST89, SB90, BALL90], and security issues [Bir85].

# 2 Properties of RPC Services

## 2.1 Point-to-point RPC

The properties of RPC can be classified into categories as follows.

- *Failure semantics* specify what guarantees are given to the client about the execution of the server procedure, both when the call returns successfully and when the call returns unsuccessfully. The two properties are *unique execution*, which states that the server procedure is not executed more than once, and *atomic execution*, which states that the server procedure is either executed completely or not at all.

- *Call semantics* specify the synchrony of the client call. A call is *synchronous* if the client thread is blocked until the call to the server is finished, while a call is *asynchronous* if the client thread

returns immediately. In the latter case, the RPC system may include another system call that allows the thread to retrieve results later. Although synchronous is most commonly used, a number of systems provide an asynchronous option as well [ATK91, LS88, WNF90].

- *Orphan handling semantics* specify how *orphans*—that is, server computations associated with clients that have failed—are dealt with. Orphans not only waste computing resources, but may also interfere with new calls issued by a recovered client. Options for dealing with orphans include *interference avoidance*, where the orphans finish their computation before the recovered client is allowed to issue new requests, and *orphan termination*, where orphans are terminated upon detection [Shr83, PS88].

- *Communication semantics* specify properties about the communication between the client and server. Here, we concentrate on *reliable communication*, which can be implemented by message acknowledgements and retransmissions. Of course, if the reliability guarantees provided by the underlying communication layer are strong enough, then the RPC layer may not need to implement reliability. Furthermore, an application builder might choose not to have this property for other reasons, e.g., efficiency or cost.

- *Termination semantics* specify the guarantees that are given about termination of a call. Due to communication and server failures, the client site may retry a call an arbitrary number of times. *Bounded termination* states that a call always terminates and the client thread returns within a bounded, specified time. If the server has not responded by the deadline, the call returns with an indication of failure.

We point out that our classification of failure semantics subsumes more traditional distinctions, which can be summarized as follows [PS88]. *At least once* guarantees that if the invocation terminates normally, the remote procedure has been executed one or more times, and if it terminates abnormally, no conclusion is possible [Spe82]. *Exactly once* guarantees that if the invocation terminates normally, the remote procedure has been executed exactly one time, and if it terminates abnormally, no conclusion is possible other than that it has not been executed more than once. *At most once* is the same as "exactly once" if the invocation terminates normally, while if the invocation terminates abnormally, the execution of the remote procedure is guaranteed to be atomic, i.e., either completely executed or not at all [LS83]. In our classification, each of these semantics can be realized as some combination of the unique and atomic execution properties, as illustrated in Figure 1.

| | Unique execution | Atomicity of procedure execution |
|---|---|---|
| At least once | NO | NO |
| Exactly once | YES | NO |
| At most once | YES | YES |

Figure 1: Failure semantics as combinations of properties

## 2.2 Group RPC

Group RPC is any RPC service where the request is sent to more than one server—i.e., a *server group*—using either multicast or point-to-point communication. Group RPC has numerous applications. For

example, it can be used to implement replicated servers to increase availability of the service in the event of failures, to implement parallel computation, or to improve response time. Examples of group or multicast RPC include [Coo85, YJT88, CGR88, WZZ93, Che86, SS90, Mar86, Coo90].

For brevity, in this paper we consider only one-to-many group RPC, in which one client uses RPC to invoke a procedure implemented by a server group. The semantics of group RPC are identical to ordinary RPC when considering the call, orphan handling, communication, and termination semantics discussed above. However, group RPC also includes the following.

- *Ordering semantics* specify the order in which concurrent calls are executed by different members of the server group. FIFO ordering guarantees that all calls issued by any one client are executed in the same order by all group members, while total order guarantees that all calls are executed in the same total order. Other variants such as partial or causal order have also been defined.
- *Collation semantics* specify how responses from the multiple members of the group are combined before being returned to the client. Different possibilities include return any reply, return all replies, or return the result of a function that maps all replies into one result (e.g., average). Of course, any of these alternatives can be described as a function, so we take the general approach of having the user provide the desired collation function at initialization time.
- *Failure semantics* are defined as above, but must be augmented given more than one server. In particular, different combinations of the failed and successful executions must be considered.

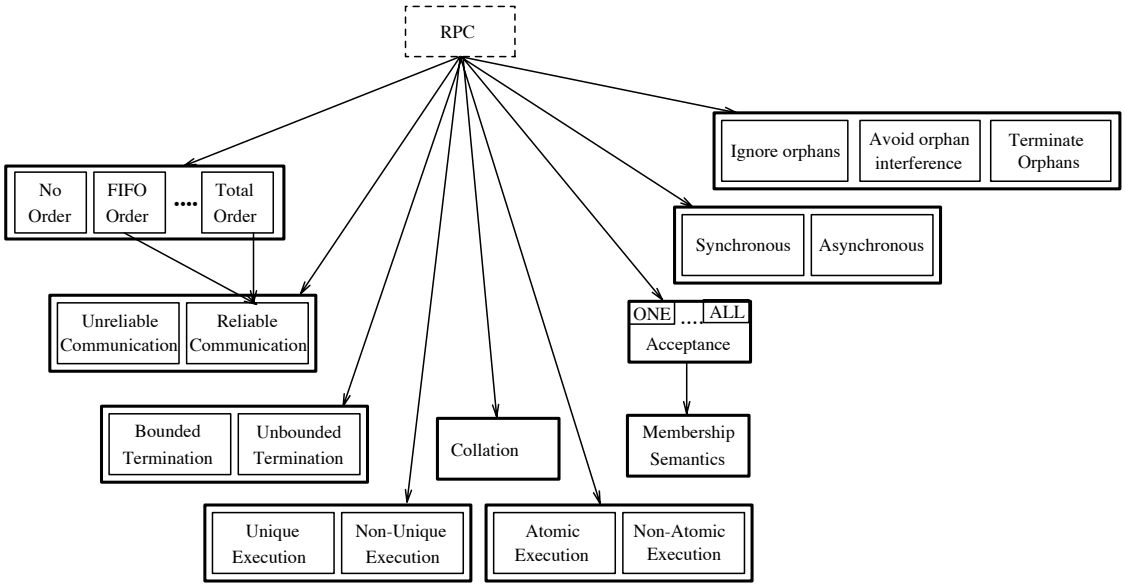


Figure 2: Semantic properties of group RPC

As noted, the failure semantics become much more complicated with multiple servers. Specifically, we must now consider how many servers must succeed in order for the group RPC to be considered successful, a property we term its *acceptance semantics*. Possibilities range from requiring successful execution at only one server to successful execution at all servers. Note that the concept of "all" is not trivial. For example, if we assume the server group has a fixed membership and that all sites eventually

recover, a response will be forthcoming from all servers if the client waits long enough. On the other hand, the client might not want to wait for recovery, but is willing to settle for the responses from all servers that are still functioning. Dealing with site failure and recovery in this way constitutes the *membership semantics*.

Figure 2 summarizes the properties of group RPC and by implication, point-to-point RPC. Each box represents a property, with alternative variants of the semantics represented as enclosed labeled boxes. The edges represent logical dependencies between properties in the sense that a property $p_1$ depends on property $p_2$ if $p_2$ must hold in order for $p_1$ to hold. For example, to implement FIFO or total ordering, every server must receive the same set of messages, i.e., the reliability property must hold.

# 3 Micro-protocols and event-driven execution

A configurable RPC system based on the above properties is realized using a model for composing fine-grained software modules [HS93] and its associated *x*-kernel based implementation platform [BS94]. The basic building block of this model is a collection of *micro-protocols*, each of which implements a well-defined property. A micro-protocol, in turn, is structured as a collection of *event handlers*, which are procedure-like segments of code that are invoked when an *event* occurs. Events can be either user or system defined, and are used to signify changes of state potentially of interest to the micro-protocol. For example, a commonly-used event for building network protocols like RPC is "message arrival." When an event is detected, all event handlers registered for that event are invoked; events can also be generated explicitly by micro-protocols, with the same effect. The invocation of event handlers due to the occurrence of a single event can be *sequential*—performed sequentially using one thread of control, or *concurrent*—performed concurrently with each event handler given its own thread of control. The invocation itself can be *blocking*, where the invoker waits until all the event handlers registered for the event have finished execution, or *non-blocking*, where the invoker continues execution without waiting.

Event registration, detection, and invocation are implemented by a standard runtime or *framework* that is linked with the micro-protocols. The framework also supports shared data (e.g., messages) that can be accessed by the micro-protocols configured into the framework. The object formed by the linking of a collection of micro-protocols and associated framework is known as a *composite protocol*. Once created, such a composite protocol can be composed in a traditional hierarchical manner with other *x*-kernel protocols to form the application's protocol stack. To accomplish this, a composite protocol exports the standard *x*-kernel Uniform Protocol Interface (UPI), even though its internal structure is richer than a standard *x*-kernel protocol.

An example composite protocol is depicted in Figure 3. In the middle is the framework, which contains a shared data structure—in this case a table of pending RPC calls—and some event definitions. The boxes to the left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs.

The following operations are provided to micro-protocols by the framework for dealing with events.

- **register**(*event_name*,*event_handler_name*, *priority*), which is used to request that the framework invoke handler *event_handler_name* when *event_name* occurs. If the event is sequential, the event handlers registered for the event are executed in priority order based on the *priority* value each supplied when they registered. If omitted, the value defaults to the lowest priority.

- **trigger**(*event_name*,*arguments*), which is used to notify the framework that event *event_name* has occurred. The framework will then execute all the event handlers registered for this event, passing *arguments* in the invocation.

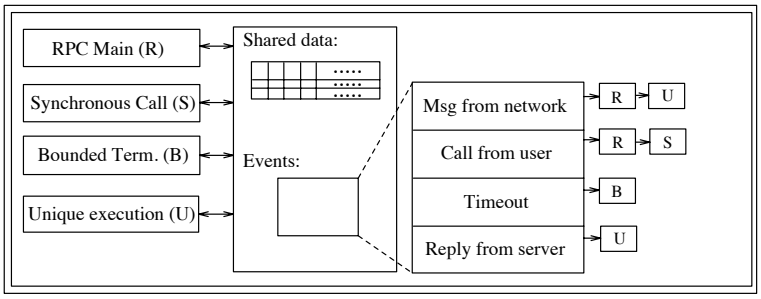


Figure 3: A composite protocol

- **deregister**(*event_name*,*event_handler_name*), which is used to reverse the registration process.
- **cancel_event()**, which is used to notify the framework that the current event is to be cancelled, i.e., the remaining event handlers registered for this event need not be executed. This operation is mostly useful for sequential events.

The model also has a provision for events triggered by the passage of time. To request this, a micro-protocol uses the **register** procedure with TIMEOUT as the event name and specifies the time interval as the priority parameter. With the exception of the TIMEOUT event, event handlers remain registered for their event until explicitly deregistered, so that each may be invoked any number of times. Event handlers registered for the TIMEOUT event are executed only once after the timeout period has expired.

# 4 Implementing a group RPC service

## 4.1 Outline

In this section, we present detailed pseudocode illustrating how RPC can be implemented in a modular and configurable fashion using the model described in the previous section. For brevity, the focus here is on group RPC. Point-to-point RPC can be seen as a special case in this implementation, although in practice it would likely be implemented separately to obtain a more compact and efficient protocol.

In following the tenets of the model, the RPC service is implemented as a composite protocol gRPC. We assume that the system also includes the following composite or simple protocols: unreliable communication, user (i.e., the server or client code), and possibly membership. The unreliable communication protocol provides the transport service needed to deliver messages between gRPC on the client and server sites. We also assume that the client above gRPC has a stub for each RPC call that marshalls arguments and does binding. A similar stub on the server site unmarshalls the data and invokes the actual procedure. From the perspective of gRPC, then, the arguments are treated as one continuous untyped field that is copied to and from messages.

## 4.2 gRPC framework

The framework contains a number of shared data structures. The first is *pRPC*, a table for storing pending remote procedure calls at the client.

```
type  Status_type =    enum{ OK, WAITING, TIMEOUT };
      waiting_list =   table of { p:process_id, acked: bool, done: bool } indexed by p;

      Client_Record = record {id:      int;          – call identifier
                              op:      op_id;        – operation identifier
                              args:    arg_type;     – input and output parameters
                              server:  group_id;     – identity of server group
                              sem:     semaphore;    – sem. where client thread waits
                              nres:    int;          – num. of responses required
                              pending: waiting_list; – response waited from
                              status:  Status_type; } – return status of call

      Client_Table =   table of Client_Record indexed by id;

var   pRPC :           Client_table;
      pRPC_mutex:      semaphore;                    – control access to pRPC
```

The call identifier, *id*, is carried along with the call to the server and its response so that calls and responses can be matched. The number of responses required field, *nres*, has a value of 1 for point-to-point RPC and a value depending on the acceptance policy for group RPC. The *pending* field lists the process identifiers of the servers that have yet to respond. The *pRPC* table is indexed using the call identifier (e.g., *pRPC(id)*), while other fields are accessed using record notation (e.g., *pRPC(id).sem*).

A similar data structure, *sRPC*, is used at the server to store information about each pending client call:

```
type  ready_index =    [1..N];                       – N is number of properties
      hold_table =     array [ready_index] of bool;

      Server_Record = record {id: int; op: op_id; args: arg_type; server: group_id;
                              client:  process_id;   – identity of client
                              hold:    hold_table; } – array of properties satisfied

      Server_Table =   table of Server_Record indexed by id;

var   sRPC :           Server_table;
      sRPC_mutex:      semaphore;                    – control access to sRPC
```

As described above, messages are exchanged between the user protocol and gRPC, and between gRPC and the underlying communication protocol. The type definitions for these messages is as follows:

```
type  Net_Optype =     enum{ Call, Reply, ACK, Order };
      Net_Msgtype =    record {type:    Net_Optype;   – type of message
                               id: int; op: op_id; args: arg_type; server: group_id;
                               sender:  process_id;   – sender of message
                               inc:     int;          – incarnation number
                               ackid:   int; }        – id of a call being acknowledged

      User_Optype =    enum{ Call, Request };
      User_Msgtype =   record {type: User_Optype; id: int; op: op_id;
                               args: arg_type; server: server_id; status: Status_type; }

      Mem_Change =     enum{ FAILURE, RECOVERY };

var   Net :            ptr Protocol;          – underlying network protocol
      Server :         ptr Protocol;          – server protocol
      inc_number:      int;                   – current incarnation nbr
      Members:         set of process_id;     – live members
      HOLD:            hold_array;            – array of properties to be satisfied
      serial:          semaphore;             – semaphore for enforcing serial execution at server
```

Here, variable *Net* is a pointer to the communication protocol, and a point-to-point send or multicast operation that can be executed with the *x*-kernel operation *Net.push(...)* to send messages. Variable *Server* is an analogous pointer to the user protocol, and an operation *Server.pop(...)* that is used to pass messages up the protocol stack. A call to this operation is blocking. The *HOLD* array is used to indicate which properties must be satisfied before a call can be passed up to the server, or, in other words, which micro-protocols must process the message. An analogous array is associated with each call indicating which properties have been satisfied, and when the two are equal, the call is given to the server.

## 4.3 Events

The events used by gRPC's micro-protocols are the following; for simplicity, we assume all events are blocking and sequential:

- CALL_FROM_USER(umsg:User_Msgtype): Triggered at the client side when a new call from the user protocol arrives.
- NEW_RPC_CALL(id:int): Triggered at the client side when a call is ready to exit gRPC and be sent to the server site. This event is used primarily by micro-protocols to update data structures before the invocation is sent.
- REPLY_FROM_SERVER(id:int): Triggered at the server side when the server passes the response to a call to gRPC.
- MSG_FROM_NETWORK(msg: Net_Msgtype): Triggered when a message arrives from the network; both the client and server sides.
- RECOVERY(inc_number:int): Triggered when failed site is recovering; both client and server sides. The argument *inc_number* is the sequence number of the current incarnation.
- MEMBERSHIP_CHANGE(who: process_id, change: Mem_Change): Triggered by the membership service when a process fails or recovers. Most properties identified in section 2 do not require this information in their implementations, so the membership component of the system is omitted in these cases.

## 4.4 RPC micro-protocols

### 4.4.1 RPC Main

The *RPC_Main* micro protocol handles the main control flow of an RPC on both the client and server sides. Specifically, it stores the call request in *pRPC*, sends the request over the network, issues the call to the server, sends the response over the network, and stores the results in *pRPC*. Note that this micro-protocol does not handle blocking of user threads.

```
micro-protocol RPC_Main(Net, Server: ptr Protocol)
    var next_id: int;

    event handler msg_from_net(msg: Net_Msgtype)
    var rec: Server_Record;
    begin
        if msg.type = Call then rec = { msg.id, msg.op, msg.args, msg.server, msg.sender, };
            rec.hold[] = false; sRPC += rec; forward_up(msg.id,MAIN); end
    end
```

```
    export procedure forward_up(id: int, index: ready_index)
    var execute = true;
        msg: Net_Msgtype;
    begin
        sRPC(id).hold[index] = true;
        for each i: ready_index do
            if HOLD[i] and not sRPC(id).hold[i] then execute = false; end
        end
        if execute then
            Server.pop(sRPC(id).op,sRPC(id).args); trigger(REPLY_FROM_SERVER,id);
            msg = { Reply, id, sRPC(id).op, sRPC(id).args, sRPC(id).server, my_id, , };
            sRPC -= sRPC(id); Net.push(sRPC(id).client,msg);
        end
    end

    event handler msg_from_user(umsg:User_Msgtype)
    var msg: Net_Msgtype;
        rec: Client_Record;
    begin
        if umsg.type = Call then
            P(pRPC_mutex);
            rec = { next_id, umsg.op, umsg.args, umsg.server, 0, (waiting_list)umsg.server, WAITING };
            next_id++; pRPC += rec; V(pRPC_mutex); trigger(NEW_RPC_CALL,rec.id); umsg.id = rec.id;
            msg = { Call, rec.id, rec.op, rec.args, my_id, umsg.server, inc_number, };
            Net.push(rec.server,msg);
        end
    end

    event handler handle_recovery(inc:int) begin inc_number = inc; end

    register(MSG_FROM_NETWORK,msg_from_net,3); HOLD[MAIN] = true;
    register(CALL_FROM_USER,msg_from_user,1); register(RECOVERY,handle_recovery);
end RPC_Main
```

In the discussion of the *HOLD* array, we noted that the *hold* arrays of individual calls are compared with *HOLD* and when all properties are satisfied, the request is forwarded to the server. This checking and eventual forwarding is done by calling the *forward_up* procedure, which is exported by the *RPC_Main* micro-protocol.

### 4.4.2 User thread management

As described in section 2, an RPC invocation can be either synchronous (blocking) or asynchronous (non-blocking). The *Synchronous_Call* micro-protocol implements synchronous RPC semantics by blocking the caller thread and matching responses with pending threads.

```
micro-protocol Synchronous_Call()

    event handler msg_from_user(umsg:User_Msgtype)
    begin
        if umsg.type = Call then
            P(pRPC(umsg.id).sem); umsg.args = pRPC(umsg.id).args; umsg.status = pRPC(umsg.id).status;
            P(pRPC_mutex); pRPC -= pRPC(umsg.id); V(pRPC_mutex);
        end
    end

    register(CALL_FROM_USER,msg_from_user);
end Synchronous_Call
```

*Asynchronous_Call* implements a very simple asynchronous RPC where the caller thread is not blocked when the call is issued, but may later request the result using a *Request* message. If the result is pending, the request message returns immediately, otherwise, the caller is blocked until the result arrives or the call is otherwise terminated.

```
micro-protocol Asynchronous_Call()

    event handler msg_from_user(umsg:User_Msgtype)
    begin
        if umsg.type = Request then
            P(pRPC(umsg.id).sem); umsg.args = pRPC(umsg.id).args; umsg.status = pRPC(umsg.id).status;
            P(pRPC_mutex); pRPC -= pRPC(umsg.id); V(pRPC_mutex);
        end
    end

    register(CALL_FROM_USER,msg_from_user);
end Asynchronous_Call
```

### 4.4.3 Communication aspects

The standard approach to making RPC reliable is to retransmit the call to the server site until the response or some other form of acknowledgment arrives. The *Reliable_Communication* micro-protocol implements these retransmissions and acknowledgements. The *pRPC.pending(id).acked* field is used to keep track of if the call has been acknowledged or not.

```
micro-protocol Reliable_Communication(retrans_timeout: real)

    event handler handle_new_call(id:int)
    begin
        for each p:process_id in pRPC(id).pending do pRPC(id).pending(p).acked = false; end
    end

    event handler msg_from_net(msg: Net_Msgtype)
    var client: process_id;
    begin
        if msg.type = Reply and exists pRPC(msg.id) then pRPC(msg.id).pending(msg.sender).acked = true;
        elsif msg.type = ACK and exists pRPC(msg.ackid) then
            pRPC(msg.ackid).pending(msg.sender).acked = true; end
    end

    event handler handle_timeout()
    var msg = new(Net_Msgtype);
    begin
        for each id:int in pRPC do
            for each p:process_id in pRPC(id).pending do
                if not pRPC(id).pending(p).acked then
                    msg = { Call, id, pRPC(id).op, pRPC(id).args, pRPC(id).server, my_id, inc_number, };
                    Net.push(p,msg);
                end
            end
        end
        register(TIMEOUT,handle_timeout,retrans_timeout);
    end

    register(MSG_FROM_NETWORK,msg_from_net,1); register(NEW_RPC_CALL,handle_new_call);
    register(TIMEOUT,handle_timeout,retrans_timeout);
end Reliable_Communication
```

*RPC_Main* combined with *Reliable_Communication* provides for unbounded termination, i.e., the gRPC protocol at the client side keeps on trying until it gets a response. In order to guarantee bounded termination, either a limit of the amount of time that can pass or the number of retransmissions can be used. The following implementation of *Bounded_Termination* micro-protocol uses a timebound. A queue *Calls* is used to store pending calls.

```
micro-protocol Bounded_Termination(timebound: real)
    var Calls: queue of int;

    event handler handle_new_call(id:int)
    begin enqueue(Calls,id); register(TIMEOUT,handle_timeout,timebound);
    end

    event handler handle_timeout()
    var id: int;
    begin
        id = dequeue(Calls); P(pRPC_mutex);
        if exists pRPC(id) then pRPC(id).status = TIMEOUT; V(pRPC(id).sem); end
        V(pRPC_mutex);
    end

    register(NEW_RPC_CALL,handle_new_call);
end Bounded_Termination
```

### 4.4.4 Response handling

The *Collation* micro-protocol implements collation semantics, taking the function from the user protocol as a parameter.

```
micro-protocol Collation(cum_func:function,init:arg_type)

    event handler msg_from_net(msg: Net_Msgtype)
    begin
        if msg.type = Reply and exists pRPC(msg.id) then
            P(pRPC_mutex);
            pRPC(msg.id).args = cum_func(pRPC(msg.id).args,msg.args);
            V(pRPC_mutex);
        end
    end

    event handler handle_new_call(id:int)
    begin pRPC(id).args = init; end

    register(MSG_FROM_NETWORK,msg_from_net,4);
    register(NEW_RPC_CALL,handle_new_call);
end Collation
```

### 4.4.5 Failure semantics

*RPC_Main* and *Reliable_Communication* combined with *Synchronous_Call* or *Asynchronous_Call* provide the equivalent of "at least once" semantics. To implement "exactly once semantics," gRPC must guarantee that a call will not be executed more than once at each server, i.e., the unique execution property from section 2. The basic strategy is to keep track of requests that have already been executed. In our solution, the server stores its response to the original request until the client acknowledges the response. If a duplicate request is received after the acknowledgement has been received, the message is assumed to be old and simply discarded. To keep track of which calls have already been executed and their results, the micro-protocol on the server side uses data structures *OldCalls* and *OldResults*, respectively.

```
micro-protocol Unique_Execution()
    var OldCalls: set of int;
        OldResults: table of {id: int, args: arg_type} indexed by id;

    event handler handle_reply(id:int)
    begin OldResults += { id, sRPC(id).args }; end
```

```
    event handler msg_from_net(msg: Net_Msgtype)
    var new_msg: Net_Msgtype;
    begin
        if msg.type = Call then
            if exists OldResults(msg.id) then
                new_msg = {Reply, msg.id, msg.op, OldResults(msg.id).args, msg.server, my_id, inc_number,}
                Net.push(msg.sender,new_msg); cancel_event();
            elsif exists OldCalls(msg.id) then cancel_event();
            else OldCalls += msg.id; end
        elsif msg.type = Reply then
            new_msg = { ACK, , , , msg.server, my_id, inc_number, msg.id };
            Net.push(msg.sender, new_msg);
        elsif msg.type = ACK then OldResults -= OldResults(msg.ackid); end
    end

    register(MSG_FROM_NETWORK,msg_from_net,2); register(REPLY_FROM_SERVER,handle_reply,1);
end Unique_Execution
```

To provide "at most once" semantics, gRPC also has to guarantee that execution of the server procedure is atomic, i.e., the atomicity property from section 2. In situations where the server has no *stable state*—that is, state that would persist across failures, such as values stored on disk—execution is automatically atomic. On the other hand, if the server does have stable state, transactional techniques must be used to guarantee atomicity. These techniques can either be implemented in the server itself, or, with some extra support, within the RPC layer. The tradeoff is efficiency versus transparency: implementing the atomicity within the server means that the technique used can be more application specific, while doing it within the RPC layer simplifies the task of programming the server at the cost of some execution overhead.

Here, we outline an *Atomic_Execution* micro-protocol that follows the second approach of implementing atomicity in the RPC layer. To support this, the micro-protocol must have the ability to write a checkpoint of the (volatile and stable) state of the server to stable storage. The operation *checkpoint()* is assumed to write such a checkpoint and return the address of the storage location. An analogous operation *load(address)* is used to restart the server from the checkpoint stored at location *address*. In the following, simple variables that reside in non-volatile storage are labeled **stable**; assignment to these variables is assumed to be atomic.

```
micro-protocol Atomic_Execution()
    var old, new: stable address; – addr. of checkpoints

    event handler handle_reply(id:int)
    begin new = checkpoint(); old = new; end

    event handler handle_recovery(inc:int)
    begin sRPC_mutex = 0; load(old); V(sRPC_mutex); end

    register(REPLY_FROM_SERVER,handle_reply,2);
    register(RECOVERY,handle_recovery);
end Atomic_Execution
```

Note that this implementation is inefficient when the state of the user protocol is large. This can be optimized by just storing the changes ("deltas") from one checkpoint to the next. Other techniques can be found in, for example, [BHG87].

This technique for atomic execution only works if calls are processed one at a time by the server, so an additional micro-protocol, *Serial_Execution*, is also needed.

```
micro-protocol Serial_Execution()

    event handler msg_from_net(msg: Net_Msgtype)
    begin if msg.type = Call then P(serial); end end

    event handler handle_reply(id:int)
    begin V(serial); end

    serial = 1; register(MSG_FROM_NETWORK,msg_from_net);
    register(REPLY_FROM_SERVER,handle_reply);
end Serial_Execution
```

The *Acceptance* micro-protocol implements the corresponding property. In order for a call to be accepted, it must be executed successfully by at least *Acceptance_Limit* members of the server group, where *Acceptance_Limit* is specified as a parameter at initialization time. If the acceptance limit is greater than the number of group members, the number of required responses is set to the size of the group. Note that, if no membership service is available, the event MEMBERSHIP_CHANGE will never be triggered and the set "Members" will remain constant. This means that a call will only terminate when *Acceptance_Limit* responses are received even when some servers fail, or, in case of bounded termination, when the time limit expires.

```
micro-protocol Acceptance(Acceptance_Limit:int)

    event handler handle_new_call(id:int)
    var alive: int = 0;
    begin
        for each p:process_id in pRPC(id).pending do
            if p in Members then pRPC(id).pending(p).done = false; alive++;
            else pRPC(id).pending(p).done = true; end
        end
        pRPC(id).nres = min(Acceptance_Limit,alive);
    end

    event handler msg_from_net(msg: Net_Msgtype)
    begin
        if msg.type = Reply then
            if exists pRPC(msg.id) and not pRPC(msg.id).pending(msg.sender).done then
                pRPC(msg.id).pending(msg.sender).done = true; pRPC(msg.id).nres–;
                if pRPC(msg.id).nres <= 0 then pRPC(msg.id).status = OK; V(pRPC(msg.id).sem); end
            else cancel_event(); end
        end
    end

    event handler server_failure(who: process_id, change: Mem_Change)
    begin
        if change = FAILURE then
            for each id:int in pRPC do
                if exists pRPC(id) and not pRPC(id).pending(who).done then
                    pRPC(id).pending(who).done = true; pRPC(id).nres–;
                    if pRPC(id).nres <= 0 then pRPC(id).status = OK; V(pRPC(id).sem); end
                end
            end
        end
    end

    register(NEW_RPC_CALL,handle_new_call); register(MEMBERSHIP_CHANGE,server_failure);
    register(MSG_FROM_NETWORK,msg_from_net,3);
end Acceptance
```

### 4.4.6 Ordering calls

The default execution order of the client calls at the server group members is entirely arbitrary, even to the point where calls from the same client may be executed in a different order by different servers. Restricting the order is, however, straightforward by incorporating the suitable micro-protocol.

Two micro-protocols for ordering have been defined: *FIFO_Order* and *Total_Order*. *FIFO_Order* guarantees that the calls from each client will be served in a FIFO order at every server. *Total_Order*, on the other hand, guarantees that calls from all clients are processed in the same order by all servers.

```
micro-protocol FIFO_Order()
    var In_Progress: table of {client: process_id, inc: int, next: int } indexed by client;
        mutex: semaphore;

    event handler msg_from_net(msg: Net_Msgtype)
    begin
        if msg.type = Call then
            P(mutex);
            if exists In_Progress(msg.sender) then
                if In_Progress(msg.sender).inc > msg.inc or msg.id < In_Progress(msg.sender).next then
                    cancel_event(); sRPC -= SRPC(msg.id); V(mutex); exit();
                elseif In_Progress(msg.sender).inc < msg.inc then
                    In_Progress(msg.sender) = { msg.sender, msg.inc, msg.id }; end
            else In_Progress += { msg.sender, msg.inc, msg.id }; end
            V(mutex);
            if msg.id = In_Progress(msg.sender).next then forward_up(msg.id,FIFO); end
        end
    end

    event handler handle_reply(id:int)
    begin
        P(mutex); In_Progress(sRPC(id).client).next = id + 1; V(mutex);
        if exists sRPC(id+1) then forward_up(id+1,FIFO); end
    end

    register(MSG_FROM_NETWORK,msg_from_net,10);
    register(REPLY_FROM_SERVER,handle_reply,1); HOLD[FIFO] = true;
end FIFO_Order
```

Notice that *FIFO_Order* has been deliberately written so that it allows possible duplicate execution of a call and it allows concurrent execution.

The *Total_Order* micro-protocol uses one group member, the *leader*, to assign the total order in which calls will be executed and then disseminate it to the group. The leader at any point is defined to be the server with the largest unique identifier of all non-failed servers. Thus, for example, if the initial leader fails, the server with second largest identifier takes over. The solution presented below is slightly simplified when it comes to changing the leader. In order to guarantee consistent total order over leader changes the remaining sites should reach an agreement on what was the last ordering message sent by the leader before it failed. For brevity this agreement phase has been omitted from the code segment below.

```
micro-protocol Total_Order()
    var Ready_list: table of { id: int, order: int } indexed by order;
        Waiting_set: table of { id: int } indexed by id;
        next_order: int;                    – next order to be assigned to a request
        leader_mutex: semaphore;            – control access to next_order
        next_entry: int;                    – the order that is allowed to enter
        Old_Order: table of { id: int, order: int } indexed by id;

    function leader(server: group_id) returns process_id begin return(max(id: process_id in server)); end
```

```
    event handler assign_order(msg: Net_Msgtype)
    var order_msg: Net_Msgtype;
    begin
        if msg.type = Call then
            if my_id = leader(msg.server) then
                order_msg = { Order, msg.id, , , , my_id, inc_number, }; P(leader_mutex);
                if exists Old_Orders(msg.id) then order_msg.ackid = Old_Orders(msg.id).order;
                else order_msg.ackid = next_order; Old_Orders += {msg.id, next_order}; next_order++; end
                V(leader_mutex); Net.push(msg.server,order_msg);
            elseif exists Waiting_set(msg.id) then Net.push(leader(msg.server),msg); end
            if exists Old_Orders(msg.id) and Old_Orders(msg.id) < next_entry then
                cancel_event(); exit(); end
        end
    end

    event handler msg_from_net(msg: Net_Msgtype)
    var my_order: int;
    begin
        if msg.type = Call then
            if not exists Old_Orders(msg.id) then Waiting_set += msg.id;
            else my_order = Old_Orders(msg.id);
                if my_order < next_entry then cancel_event(); sRPC -= sRPC(msg.id); exit();
                elseif my_order = next_entry then forward_up(msg.id,TOTAL);
                else Ready_list += { msg.id, my_order }; end
            end
        elseif msg.type = Order then
            if next_order < msg.ack_id + 1 then next_order = msg.ack_id + 1; end
            if not exists Old_Orders(msg.id) then Old_Orders += { msg.id, msg.ack_id }; end
            if exists Waiting_set(msg.id) then
                Ready_list += { msg.id, msg.ack_id }; Waiting_set -= Waiting_set(msg.id);
                if msg.ack_id = next_entry then
                    next_id = Ready_list(next_entry).id; Ready_list -= Ready_list(next_entry);
                    forward_up(next_id,TOTAL);
                end
            end
        end
    end

    event handler handle_reply(id:int)
    var next_id : int;
    begin
        next_entry++;
        if exists Ready_list(next_entry) then
            next_id = Ready_list(next_entry).id; Ready_list -= Ready_list(next_entry);
            forward_up(next_id,TOTAL);
        end
    end

    register(MSG_FROM_NETWORK,assign_order,1); HOLD[TOTAL] = true;
    register(MSG_FROM_NETWORK,msg_from_net,4); next_order = 1;
    register(REPLY_FROM_SERVER,handle_reply,1); next_entry = 1;
end Total_Order
```

This implementation of total execution order assumes that micro-protocols for reliable communication and unique execution are present and that micro-protocol for bounded termination is not present.

### 4.4.7 Dealing with orphans

The basic set of micro-protocols presented so far ignore orphans in the sense that any responses generated by orphan computations are simply ignored. This approach may, however, cause problems. For example, a client may issue a request, fail, recover, and issue the request again while the previous request is still being processed by the server. As described in section 2, two ways of dealing with these

problems are interference avoidance and orphan termination.

The micro-protocol *Interference_Avoidance* implements the first option. The solution technique is based on using client incarnation numbers to partition calls into generations. In particular, if a call from the client arrives with a new incarnation number, execution of the requested procedure can only be initiated once execution of any pending calls with old incarnation numbers have been completed. Rather than storing these calls with new numbers, here we use the approach of simply dropping them until all current calls have been finished, relying on retransmission from the client to ensure they will eventually be executed. To avoid starvation, no more calls with the old incarnation number are started once the first one with a new number has been seen.

```
micro-protocol Interference_Avoidance()
    var  Cinfo: table of {client: process_id, inc: int, count: int, next_inc: int} indexed by client;
         Cmutex: semaphore;

    event handler msg_from_net(msg: Net_Msgtype)
    var client: process_id;
    begin
        if msg.type = Call then
            client = msg.sender; P(Cmutex);
            if not exists Cinfo(client) then Cinfo += { client, msg.inc, 0, msg.inc }; end
            if Cinfo(client).inc > msg.inc then cancel_event();
            elsif Cinfo(client).inc < msg.inc then
                Cinfo(client).inc = MAX_INT; Cinfo(client).next_inc = msg.inc;
                if Cinfo(client).count = 0 then Cinfo(client).inc = msg.inc; end
            end
            if Cinfo(client).inc = msg.inc then Cinfo(client).count++; end
            V(Cmutex);
        end
    end

    event handler handle_reply(id:int)
    var client: process_id;
    begin
        P(Cmutex); client = sRPC(id).client; Cinfo(client).count–;
        if Cinfo(client).count = 0 and Cinfo(client).inc = MAX_INT then
            Cinfo(client).inc = Cinfo(client).next_inc;
        end
        V(Cmutex);
    end

    register(MSG_FROM_NETWORK,msg_from_net,2); register(REPLY_FROM_SERVER,handle_reply,1);
end Interference_Avoidance
```

The micro-protocol *Terminate_Orphan* implements the second option of immediately killing orphans as soon as they are detected. Detection can be based either on receiving a message from a newer incarnation of the client, indicating that the previous incarnation died, or by periodically probing the client. *Terminate_Orphan* uses the first approach. In order to be able to kill the orphans we have to be able to access the threads executing the server procedure and to kill those threads. For this purpose we assume operation *my_thread()* returns the thread identifier of the current thread and operation *kill(thread)* kills the thread with identifier *thread*.

```
micro-protocol Terminate_Orphan()
    var Cinfo: table of { client : process_id; inc_nbr: int; threads: list of thread; } indexed by client;
        Cmutex: semaphore;

    event handler handle_reply(id:int)
    begin
        P(Cmutex); Cinfo(sRPC(id).client).threads -= my_thread(); V(Cmutex);
    end
```

```
    event handler msg_from_net(msg: Net_Msgtype)
    var client: process_id;
    begin
        if msg.type = Call then
            client = msg.sender; P(Cmutex);
            if not exists Cinfo(client) then Cinfo += { client, msg.inc, }; end
            if Cinfo(client).inc_nbr = msg.inc then Cinfo(client).threads += my_thread();
            elsif Cinfo(client).inc_nbr > msg.inc then cancel_event();
            elsif Cinfo(client).inc_nbr < msg.inc then
                for each th:thread in Cinfo(client).threads do kill(th); V(serial); end
                Cinfo(client) = { client, msg.inc, my_thread() };
            end
            V(Cmutex);
        end
    end

    register(MSG_FROM_NETWORK,msg_from_net,2); register(REPLY_FROM_SERVER,handle_reply,1);
end Terminate_Orphan
```

# 5 Configuring a group RPC service

A group RPC service is configured by choosing those micro-protocols implementing the desired properties and then combining with the gRPC framework to form a composite protocol. The micro-protocols are not entirely independent, however, so one has to pay attention to dependencies between micro-protocols while performing the configuration. Figure 4 shows the *dependency graph* illustrating the relationship between micro-protocols. A dependency between micro-protocols $p_1$ and $p_2$ is depicted by an arrow from $p_1$ to $p_2$. The bold boxes enclosing micro-protocols indicate choice, i.e., any one, but only one, of the enclosed micro-protocols may be chosen. The group of micro-protocols surrounded by dashed line represents the minimal set of micro-protocols required for a functional system.

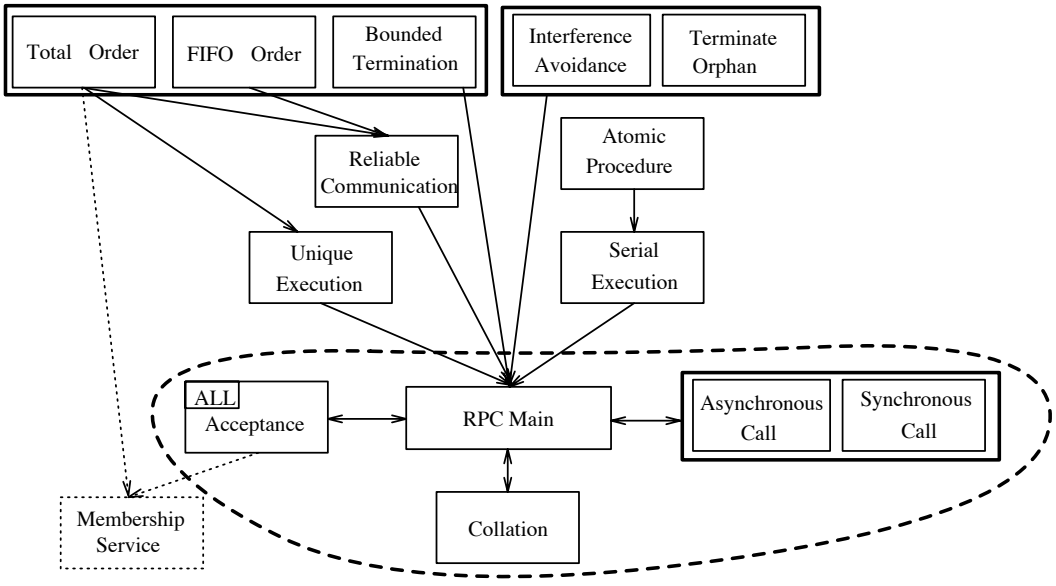


Figure 4: Dependency graph of group RPC micro-protocols

Given this graph, it is possible to determine exactly how many possible RPC services can be built with various combinations of micro-protocols. For fairness, we first fix acceptance and collation policies, since for a group of $N$ servers, there are $N + 1$ possible acceptance policies and an infinite number of possible collation policies. Even with these factors eliminated, however, micro-protocols can be selected from among two that implement different call semantics; three that deal with orphans; three that give serial execution, atomic execution, or no special execution property; and a total of 11 possible choices for dealing with unique execution, reliable communication, termination, and ordering. This sums up to $2 * 3 * 3 * 11 = 198$ possible combinations, and hence, possible group RPC services.

Also, note that the dependency graph given here does not map directly to the dependency graph of the properties given in Figure 2. One difference is that the properties in Figure 2 include those that result when something is not enforced, e.g., non-atomic execution. In an implementation, these are realized by simply not configuring in the appropriate micro-protocol, which implies they do not appear in Figure 4. The other difference is that Figure 4 contains extra dependencies that simplify the implementation rather than being inherent to the properties themselves. For example, there is an edge from *Total_Order* to *Unique_Execution* since our implementation of *Total_Order* assumes that any request is received at the server only once.

To illustrate how a specific instance of a group RPC service might be configured, consider a simple group RPC designed to provide quick response time to read-only requests. To achieve this, the system is configured with "at least once" semantics, acceptance one (i.e., only one response required), synchronous call semantics, and bounded termination time. Furthermore, we choose to implement reliability directly in the RPC service rather than relying on the underlying transport. The pseudocode for the high-level composite realizing this combination of semantics is as follows, where "||" stands for parallel composition.

```
protocol RPC_Service(Net,Server: ptr protocol)
... global type definitions ...
function id(a,b:arg_type) returns arg_type
begin return b; end

begin
    RPC_main(Net,Server) || Synchronous_Call() || Reliable_Communication(timeout) ||
    Bounded_Termination(1.0) || Collation(id,0) || Acceptance(1);
end RPC_Service
```

RPC services implementing other combinations of properties are similarly easy to construct.

# 6 Conclusions

This paper has presented an approach to constructing configurable group RPC services from a small set of micro-protocols implementing individual semantic properties. The micro-protocols are written using an event-driven execution style, and then configured with a framework that implements event detection, handler invocation, and shared variables. This results in an *x*-kernel compatible composite protocol that can be combined with others in a normal hierarchical fashion to build a network subsystem. Although the event-driven execution model is somewhat unusual, our experience is that it decouples the micro-protocols enough to facilitate configurability without adversely affecting programmability. All this derives from our experience with the Consul system, which suggests that current techniques are inadequate for supporting modularity in the type of complex protocols often found in fault-tolerant distributed systems [MPS93] .

Other researchers have also proposed modular implementations of RPC. For example, in [HPOA89] a modularization technique also based on the *x*-kernel is described. In contrast with our emphasis on modularizing along the lines of abstract properties, however, that paper describes a more syntactic scheme based on functional components of an RPC service implementing one chosen semantics. The work on agent synthesis system for Cross-RPC communication in [HR94] is relatively closely related to our goals. Although the primary goal in [HR94] is to allow heterogeneous RPC systems to communicate with one another, the system also offers the possibility for designing and prototyping new variants of RPC. In this paper, the authors divide RPC semantics into three components: call semantics (synchronous versus asynchronous), failure semantics, and RPC topology (one server versus multicast RPC). An RPC agent is synthesized from a specification written in Cicero, an event-driven specification language. Our approach to building composite protocols from micro-protocols provides more structuring support, however, and promotes a style in which RPC services are configured from a collection of already-written micro-protocols rather than generated from specifications.

## References


[AGH+91] H-R. Aschmann, N. Giger, E. Hoepli, P. Janak, and H. Kirrmann. Alphorn: A remote procedure call environment for fault-tolerant, heterogeneous, distributed systems. *IEEE Micro*, 11(5):16–19,60–67, Oct 1991.

[Apo89] Apollo Computer Inc. Network computing system (NCS) reference. Technical report, Apollo Computer Inc., 1989.

[ATK91] A.L. Ananda, B.H. Tay, and E.K. Koh. ASTRA — An asynchronous remote procedure call facility. In *Proceedings of the 8th International IEEE Conference on Distributed Computing Systems*, pages 172–179, Arlington, Texas, May 1991.

[BALL90] B. Bershad, T. Anderson, E. Lazokska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 6(1):37–55, Feb 1990.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[Bir85] A. D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, Feb 1985.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[BS94] N. T. Bhatti and R. D. Schlichting. Operating system support for configurable high-level protocols. Technical report, Department of Computer Science, University of Arizona, Tucson, AZ, 1994. in preparation.

[CGR88] R.F. Cmelik, N.H. Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[Che86] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM'86 Symposium on Communication Architectures and Protocols*, pages 406–415, Aug 1986.

[Coo85] E. C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78, Orcas Island, WA, 1985.

[Coo90] E. C. Cooper. Programming language support for multicast communication in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS-10)*, pages 450–457, Paris, France, 1990.

[Gib87] P.B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, Jan 1987.

[HP91] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HPOA89] N. C. Hutchinson, L. L. Peterson, S. O'Malley, and M. 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, AZ, Dec 1989.

[HR94] Y-M. Huang and C. Ravishankar. Designing an agent synthesis system for cross-RPC communication. *IEEE transactions on software engineering*, 19(3):188–198, Mar 1994.

[HS87] R. Hayes and R.D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transaction on Software Engineering*, 13(12):1254–1264, Dec 1987.

[HS93] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, USA, Oct 1993.

[JZ93] D. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software: practice & experience*, 23(2):201–222, 1993.

[LS83] B. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, Jul 1983.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–268, Jun 1988.

[LT91] H. Levy and E. Tempero. Modules, objects and distributed programming: Issues in RPC and remote object invocation. *Software, practice & experience*, 21(1):77–90, Jan 1991.

[Mar86] B. Martin. Parallel remote procedure call language reference and user's guide. Technical report, Computer Systems Research Group, University of California, San Diego, 1986.

[MPS93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. *Software Practice & Experience*, 23(10):1059–1075, Oct 1993.

[Nel81] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[PA88] M. Pucci and J. Alberi. Optimized communication in an extended remote procedure call model. *Computer architecture news*, 16(4):37–44, Sep 1988.

[PS88] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, SE-14(1):30–37, Jan 1988.

[RST89] R. Van Renesse, H. Van Staveren, and A. S. Tanenbaum. Performance of the Amoeba distributed operating system. *Software – Practice and Experience*, 19:223–234, Mar 1989.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 6(1):1–17, Feb 1990.

[Shr83] S. K. Shrivastava. On the treatment of orphans in a distributed system. In *Proceedings of Third Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, Florida, Oct 1983.

[Spe82] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(17):246–260, Apr 1982.

[SS90] M. Satyanarayanan and E. H. Siegel. Parallel communication in a large distributed environment. *IEEE transactions on computers*, Mar 1990.

[Sto94] A. Stoyenko. SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls. *Software, practice & experience*, 24(1):27–49, Jan 1994.

[Sun88] Sun Microsystems. RPC: Remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Jun 1988.

[TB90] Y.K. Tham and S.K. Bhonsle. Retargetable stub generator for a remote procedure call facility. *Computer communications*, 13(6):323–330, Jul 1990.

[WNF90] E. Walker, P. Neves, and R. Floyd. Asynchronous remote operation execution in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS-10)*, Paris, France, May 1990. IEEE.

[WSG91] Y.-H. Wei, A. Stoyenko, and G. Goldszmidt. The design of a stub generator for heterogeneous RPC systems. *Journal of Parallel and Distributed Computing*, 11(3):188–197, Mar 1991.

[WZZ93] Wang Xingwei, Zhao Hong, and Zhu Jiakeng. GRPC: A communication cooperation mechanism in distributed systems. *Operating Systems Review*, 27(3):75–86, Jul 1993.

[Xer81] Xerox. Courier: The remote procedure call protocol. Technical Report XSIS 038112, Xerox System Integration Standard, Stamford, CT, Dec 1981.

[YJT88] K. Yap, P. Jalote, and S. Tripathi. Fault tolerant remote procedure call. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 48–54, Jun 1988.