1 RPC

The idea behind RPC is that procedure calls are extended across machines. This is a natural paradigm to follow for programmers. Issues to be addressed were naming and binding.

1.1 Naming

How do I find the server? Idea of a naming service. Some everyday naming services include:

- Yellow Pages
- Portmapper (Sun RPC)
- DNS

Naming services all resolve a (human readable) “name” into some handle. In DNS this handle is the ordered pair: \( <\text{IPaddress}, \text{Port} > \). In some languages like java, these details are encapsulated into a “proxy" object, which has methods for reading/writing the underlying information.

1.2 Signatures

The RPC server runs an export service which consists of an interface. This is simply a group of method prototypes. For example the interface for a time service would look like this:

```c
int getCurrentTime(int TimeZone)
...
int getDay()
```
1.3 Argument Passing

An RPC client invocation involves the following chain of events:

\[ NameLook up \rightarrow Method Call \rightarrow Pass Argument(s) \]

Usually arguments are passed via call-by-:

- reference
- name
- value
- copyin-copyout

RPC only uses the latter two methods. In the earlier implementations, issues such as language and architecture differences presented a challenge for argument passing, but today with IEEE standards in place these are much less of a worry.

For complex objects and pointers, serialization must occur prior to the transfer of arguments and results. Serialization is the act of converting complex data objects into their primitive equivalents. Primatives are stored in a linear structure that reflects the order of their decomposition. The act of serialization is known as marshalling. Earlier RPC implementations relied on programmers to do marshalling by hand. This was very tedious due to languages like c, which are not strongly typed. Also, the programmer would have to cater for problems such as cycles in data structures.

These days, modern language features include auto memory management and auto serialization. Tools like SOAP, XML, and CORBA all provide intermediate representations of objects containing common data that can be handled by both parties, regardless of differences in platform or operating systems.

Some possible disadvantages include:

- Hard or impossible to optimize the serialization
- Susceptable to programming mistakes. For example (unknowingly or maliciously) supplying a pointer to a very large object to the serialization method or routine.

1.4 The RPC Mechanism

On the client side, stubs are the interface between the calling function and the server. When the client makes a call, the stub is invoked. The stub does the following:

1. Create a message
2. Identify the method
3. Marshall the arguments
4. Send the message
5. Keep-alive/check-alive
6. Unmarshall the results
7. Return

On the server side, skeletons are the interface between the client and the called method/function. Servers are multithreaded and skeletons execute in the following manner:

1. Unmarshall the arguments
2. Identify and invoke the method
3. Create a message
4. Marshall the results
5. Send the results

Further details including acknowledgements are left out at this point. (Refer to RPC paper)

1.5 Dynamic Loading

Idea behind this is find the code dynamically. There is no notion of centralized database like Grapevine. All binding information is embedded within the proxy object itself.

2 Time, Clocks, and the Ordering of Events in a Distributed System

Physical clocks are not bad. Users however make mistakes. This paper deals with a fairly natural problem: there is no total ordering of events in the world. Humans don’t like dealing with inconsistencies. We do however tend to ignore “weak consistency”, which allows the view of an object in a somewhat inconsistent state. There is a limit as to how much of this we can stand.

Presented is the idea of logical ordering, without the use of physical clocks. Logical clocks are used instead. These are monotonically increasing time streams, represented by vectors of events. A given time T is represented
by $T = < t_1, t_2, ..., t_n >$ and time $T_a > T_b$ iff $\forall i T_{a,i} > T_{b,i}$ and $\exists i, j T_{a,i} > T_{b,i}$ or $T_{a,j} < T_{b,j}$. Next, is a brief description of the algorithm proposed.

First $\rightarrow$ is used to define a relation between $a$ and $b$, in which $a \rightarrow b$ means that (1) $a$ and $b$ are events in the same process and that $a$ comes before $b$. (2) $a$ is the sending of a message and $b$ is the receipt of that message. (3) If $b \rightarrow c$, then $a \rightarrow c$. Another way to put it is that it is possible for event $a$ to causally affect event $b$. Other notations include:

- $a \rightarrow b$: denotes that $a$ and $b$ are concurrent iff $b \rightarrow a$
- $a \Rightarrow b$: denotes a total ordering of events $a$ and $b$
- $a \rightarrow b$: denotes a partial ordering of events with respect to physical time.

Next, the algorithm proposes that the following conditions are met in order to grant a resource to a process:

1. A process must release a held resource before another process can obtain that resource
2. A resource is granted to a request based on the order in which that request was made.
3. Every process must release a granted resource eventually, thus guaranteeing that every request is eventually granted.

Finally, the algorithm is defined by the five rules on page 561 of the paper.

The main way in which ordering is achieved is via broadcasts, which occur upon request and receipt of a resource. Practical applications in which such an algorithm may be used include replicated databases and airline reservation software.

The major problem with the whole idea is that if one process fails, the entire protocol fails. This can be refined using a solution to the consensus problem, which is discussed next.

### 3 Impossibility of Distributed Consensus with One Faulty Process

Consider the problem of the army generals. Army A is battling army B, which is broken into two separate regimes, $B_1$ and $B_2$. The generals for each regime must send messages and acknowledgements to each other regarding the time to attack army A. The problem is that the sender of the last message never knows if that message has been received. He can only hope that it is by sending many many messages.
The paper shows that no completely asynchronous consensus protocol can tolerate even a single unannounced process death. The proof makes the crucial assumption that processing is completely asynchronous and that processes do not know about the death of another. The authors claim that their thesis also applies to a weak form of the consensus problem. A solution to the consensus problem tries to form an agreement among the fault-free members of the resource population on a quantum of information in order to maintain the performance and integrity of the system.

3.1 Consensus Protocols

A consensus protocol is an asynchronous system of \(N\) processes \((N \geq 2)\). Each process \(p\) contains an input register \(x_p\), and output register, \(y_p\), and the storage space and program counter. The values of these things make up the internal state of a process at time \(t\) and if any one of these changes, then the process is said to change state. Processes change state according to a transition function. Communication among processes is achieved by the messaging primitives send\((p, m)\) and receive\((p)\), and a message buffer which stores messages that have been sent, but not yet received. A system can have a configuration, which consists of the internal process states and the contents of the message buffer. Configurations are denoted by \(C\). A system changes its configuration when an event, \(e(p, m)\) occurs, where \(p\) is a process and \(m\) is some message intended for \(p\).

All configurations in a system are assumed to be accessible, which means that there is a sequence of events, \(\sigma\) that takes \(C\) to \(C_1\), given that \(C_1\) is the current configuration. Lemma 1 in the paper expresses that schedules \((\sigma\sigma)\) are commutative. This forms the basis of the proof of the paper's proposal.

4 Transactions

Email is a classical example of a system in which there are inconsistencies that we tolerate. However, distributed systems in general cannot tolerate inconsistencies. A mechanism is therefore needed to build such systems efficiently. This is the motivation behind a transaction.

4.1 What is a Transaction

A transaction is a mechanism that enables a piece of work to be done such that the process:

- **is atomic** \(\Rightarrow\) either all operations are performed or none are performed,

- consistent \(\Rightarrow\) execution of all transaction are, though interleaved is equivalent to serial execution of each transaction,
• isolated—Partial results are hidden from others before a successful commit, and

• durable—if a system commits, then the results of that commit are guaranteed to exist, whether or not there is a system failure.

4.2 The Two-phase Commit Protocol

The following outlines the protocol:

1. The node acting as the coordinator sends a broadcast to all participating servers. The message sent is a request to the server regarding its willingness to commit the transaction.

2. Each server responds to the coordinator, saying whether or not it can commit.

3. If all servers agree to commit, the coordinator sends a broadcast again, saying “go ahead and commit”. Otherwise, if one or more servers replies with an abort message, the coordinator sends an “abort” message to everyone.

When a server responds to the coordinator acknowledging that it can commit, then it is guaranteed that that server can commit.