Design decisions

- Procedure call
  - Rather than message passing or remote fork
- No shared address space
- Emulate local procedure call semantics as closely as possible

Procedure call chosen as major control mechanism in Mesa language used
Mesa does have fork, but decided not to use
Does not affect design greatly either way
Shared address space can be done, but difficult to integrate into programming environment and slow.
RPC is based on the client-server architecture. You have a server which accepts requests from the client. The client wants to get the result of a method which is located on the server, so the client sends out a request, the method is run on the server, and the server returns the result. RPC can be done in many different ways, in this paper the architects chose to use procedure calls to get the results, instead of passing messages around the network, or using remote forks. This RPC has a very small scope, only the method and arguments is passed, there is no shared address space *explain shared address space* Their goal was to emulate local procedure calls as closely as possible.

RPC architecture

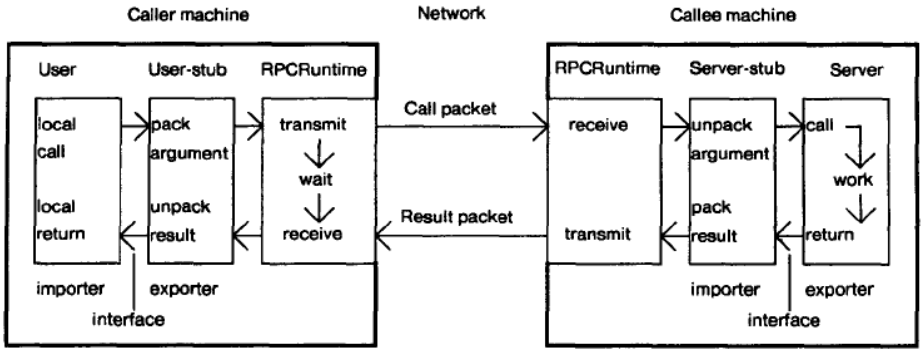

Fig. 1. The components of the system, and their interactions for a simple call.

This architecture is based on the concept of ‘stubs’. The user call gets passed to the user-stub, which is passed over the network to the server-stub which is finally passed to the server to complete the call.

A stub (aka a method-stub) is a piece of code used to stand in for some other programming functionality. It is usually used in RPC where the stub simulates a local procedure call, when really more is happening.

Service lookup / binding

- Lookup by:
  - Type (Grapevine group)
  - Instance (Grapevine individual)
  - Raw network address (not using Grapevine)

Interface identified by type (e.g. mail server) and instance (e.g. a particular mail server).

Binding made by client, stored ready for remote procedure calls. Server does not keep any state (at least not inherent in protocol) except a single sequence number

This RPC architecture uses grapevine, mainly for the reason of 'late binding'. This is where the system does not know which computer it is calling, until it needs to. This allows for the computer to not be defined at compile time. This is similar to DNS, instead of specifying an IP address, you specify a domain, this domain can be changed as needed, which allows more flexibility.

You are able to look up the 'server' which will run the procedure in three ways - you can specify the server explicitly (raw network address), name a group and get a list of instances from that group (then choose one based on a latency ordering scheme), or name an instance from the group.

Binding procedure

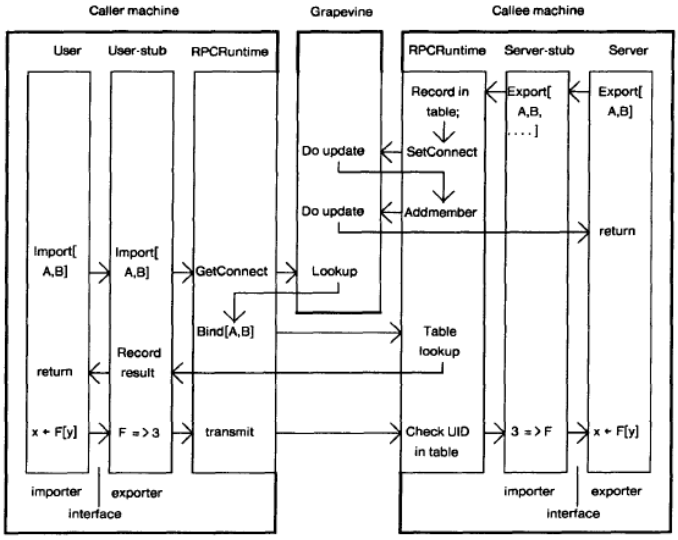

Fig. 2. The sequence of events in binding and a subsequent call. The callee machine exports the remote interface with type A and instance B. The caller machine then imports that interface. We then show the caller initiating a call to procedure F, which is the third procedure of that interface. The return is not shown.

Security

- Grapevine ACLs ensure that service instances are genuine (authorized)
- DES for end-to-end encryption, with Grapevine as KDC for end-to-end enc

ACL - Access control list
DES - Data encryption standard
KDC - Key distribution system
Security is build ontop of Grapevine's existing security architecture (which normally allows man in the middle attacks). Grapevine already ensures that the service instances are genuine (aka authorised). Dual public-key encryption can be used to ensure the messages passed are very secure.

Transport protocol

- Custom transport protocol for speed, as too much overhead in existing bytestream protocols
- Retransmissions for reliability
- Optimised for small, frequent calls
  - Calls with many packets slow, especially over high-latency connections
- No server state between calls except sequence number

Compare to TCP

No need to open or close connection: server initialises sequence number on first call received, discards it after (say) 5 minutes of inactivity as when there is no longer any danger of receiving duplicate packets

Small = arguments and return value each fit in a single packet

No need for acknowledgement packets if call is quick enough; a few are sent for calls that take a while

For longer procedure calls, probe packets are sent to ensure the callee has not crashed. After 10 minutes of probes, the probes are only sent once every 5 minutes.

For multi-packet calls, wait for acknowledgement after each packet in the call – designed for local networks & not bulk data

Simple call

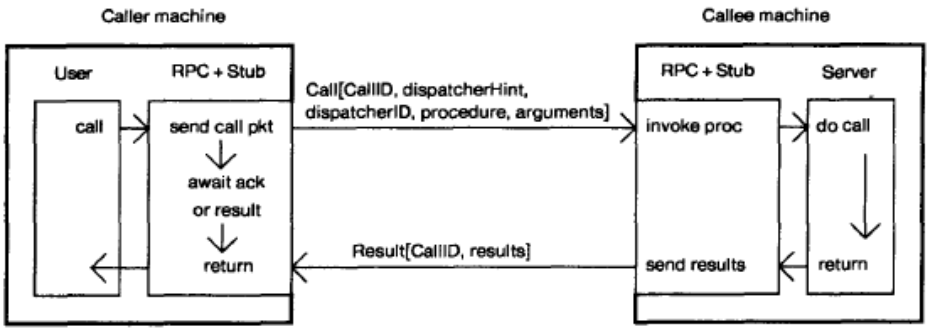

Fig. 3. The packets transmitted during a simple call.

Complicated call

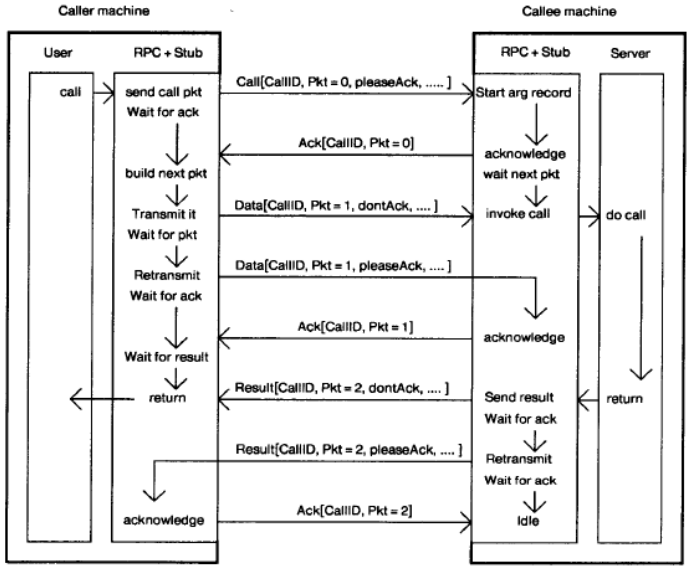

Fig. 4. A complicated call. The arguments occupy two packets. The call duration is long enough to require retransmission of the last argument packet requesting an acknowledgment, and the result packet is retransmitted requesting an acknowledgment because no subsequent call arrived.

Exception handling

- Emulate local (Mesa) exception handling almost exactly
- Extra *call failed* exception for RPC failure (network, server etc.)

Exceptions thrown by the remote procedure treated similarly to procedure calls in the opposite direction.

Performance

- Thread pool server model

Table I. Performance Results for Some Examples of Remote Calls

Procedure	Minimum	Median	Transmission	Local-only
no args/results	1059	1097	131	9
1 arg/result	1070	1105	142	10
2 args/results	1077	1127	152	11
4 args/results	1115	1171	174	12
10 args/results	1222	1278	239	17
1 word array	1069	1111	131	10
4 word array	1106	1153	174	13
10 word array	1214	1250	239	16
40 word array	1643	1695	566	51
100 word array	2915	2926	1219	98
resume except'n	2555	2637	284	134
unwind except'n	3374	3467	284	196

Discussion questions

- How does this RPC framework compare to modern frameworks?
  - Speed, overhead, efficiency
  - Client / server state
  - Ease of use
- Was it a good idea to use procedure calls as a basis rather than message passing or remote forking?
- Is it a good idea to try to emulate local procedure calls as closely as possible?

Discussion questions

- The authors discuss the possibility of a shared address space but conclude that it would be too expensive with their hardware. Is this still the case?
  - Section 1.4 paragraph 3 (page 42)
- The authors mention the possibility of using their protocol for simple calls and automatically switching to a more conventional protocol for complicated calls. Would this be worthwhile?
  - End of section 3.3 (page 54)

Discussion questions

- Is it best to have all the objects retained in memory, or can these be created/destroyed when used?
- How should procedures be passed? With pointers?
- Was it correct to use a connectionless protocol, and implement a connection on top of this? Similar to UDP instead of TCP? Resend rate?
- Should the objects be used statically, i.e. state does not matter? Should locks over objects be implemented?