Remote Procedure Calls

Taiyang Chen

10/06/2009

Overview

- Remote Procedure Call (RPC): procedure call across the network
- Lightweight Remote Procedure Call (LPRC): procedure call across domains

RPC Outline

- Background
  - History
  - Environment
- Motivation and Goals
- Design
- Implementation
  - Binding
  - Packet Transport
  - Optimizations
- Performance
- Conclusions

History

- Idea back in 1976
- Courier by Xerox in 1981
  - First business use
- Sun RPC
  - Sun Network File System in 1984
  - Now Open Network Computing RPC
  - Implementation on Unix-like systems and Windows
- A. D. Birrell and B. J. Nelson in 1984
  - Nelson's doctoral thesis

Environment

- Dorado machines (your own IBM 370/168 workstation)
- 3/10Mbps Ethernet
- Standard PUP protocol: unreliable datagram, reliable byte streams
- Cedar: programming environment for building systems and programs
- Mesa: strongly typed programming language

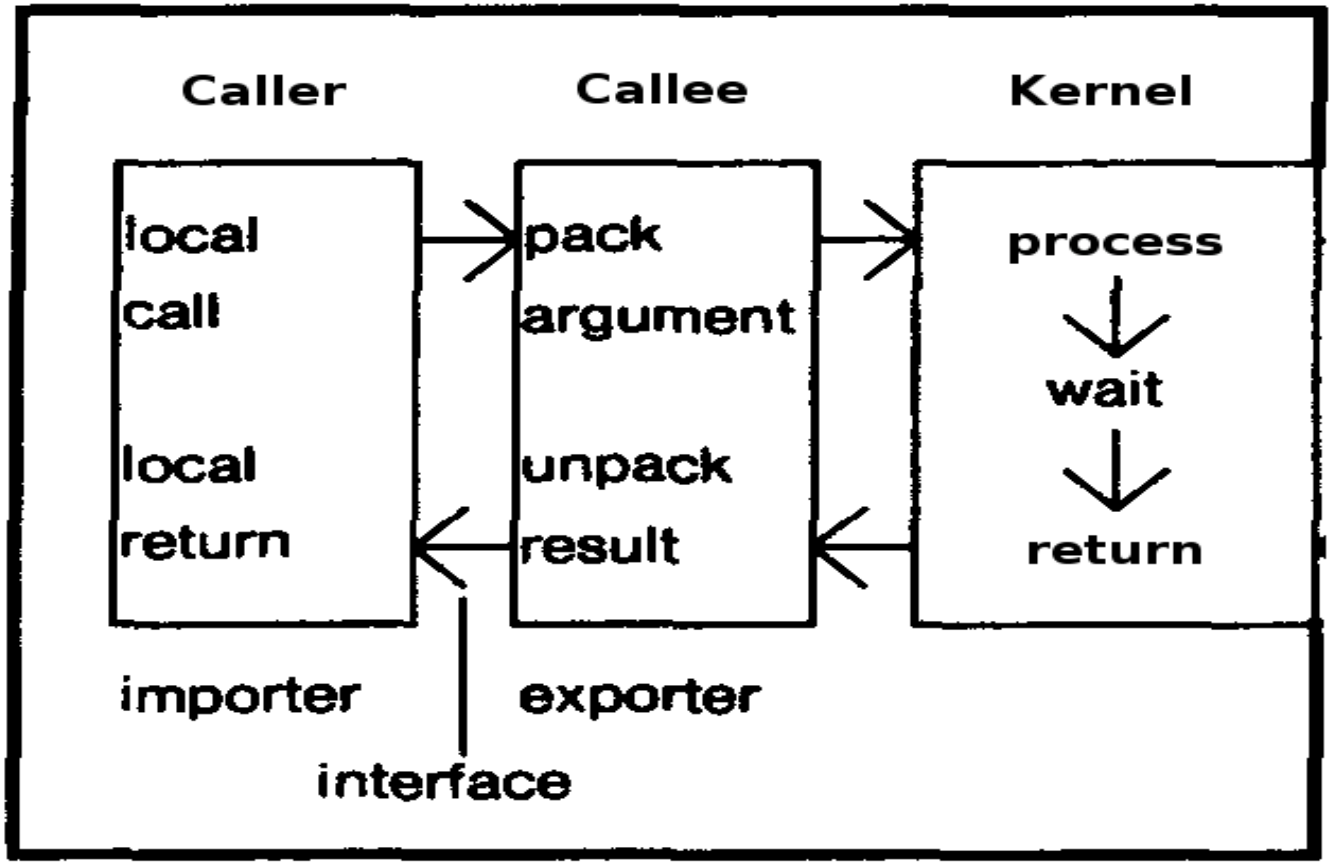
Motivation and Goals

- Distributed computing
  - Simple distributed applications using RPC
- Powerful interface
  - Ease of building RPC, like a procedure call
- Efficient calls
- Secure communication
  - Not yet, but possible

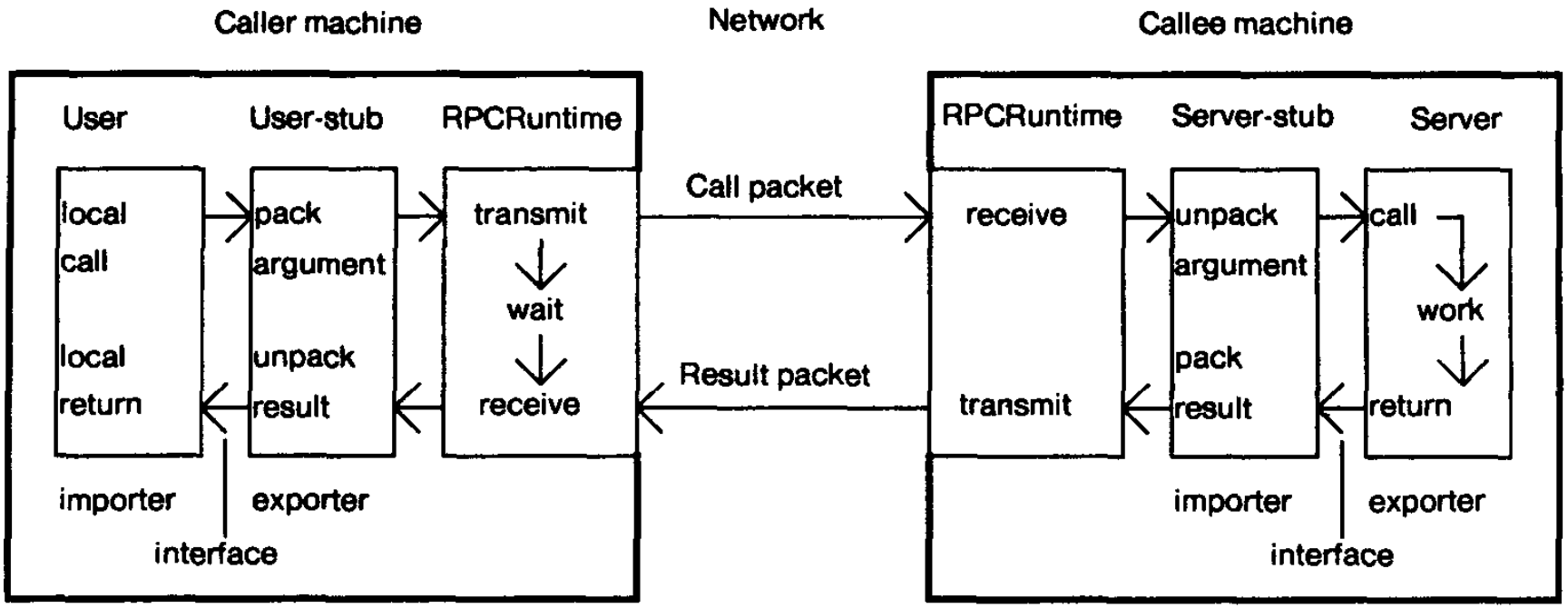
Design Decisions

- Same semantics to local procedure call
- Procedure call vs message passing
  - Reliable and efficient transmission
  - Arguments and results
  - Network security
  - Mesa
- No shared addresses
  - Paging system
  - High cost, even today

From Local Procedure Call...



To Remote Procedure Call



Components

- User/Server: caller/callee process
- Stub: packing and unpacking procedures and arguments, auto-generated by Lupine
- RPCRuntime: transport layer
- Network: PUP
- Interface: Mesa module defining procedure names, arguments and results
  - Importer
  - Exporter

Implementation

- Binding
- Packet Transport
- Optimizations

Binding

- Naming
  - Type
  - Instance
- Location
  - Grapevine: distributed database for binding
  - Key = RName
  - Entry = Individual or Group
  - Group = Set of RNames (Types)
  - Individual = Connect-site (Instance)

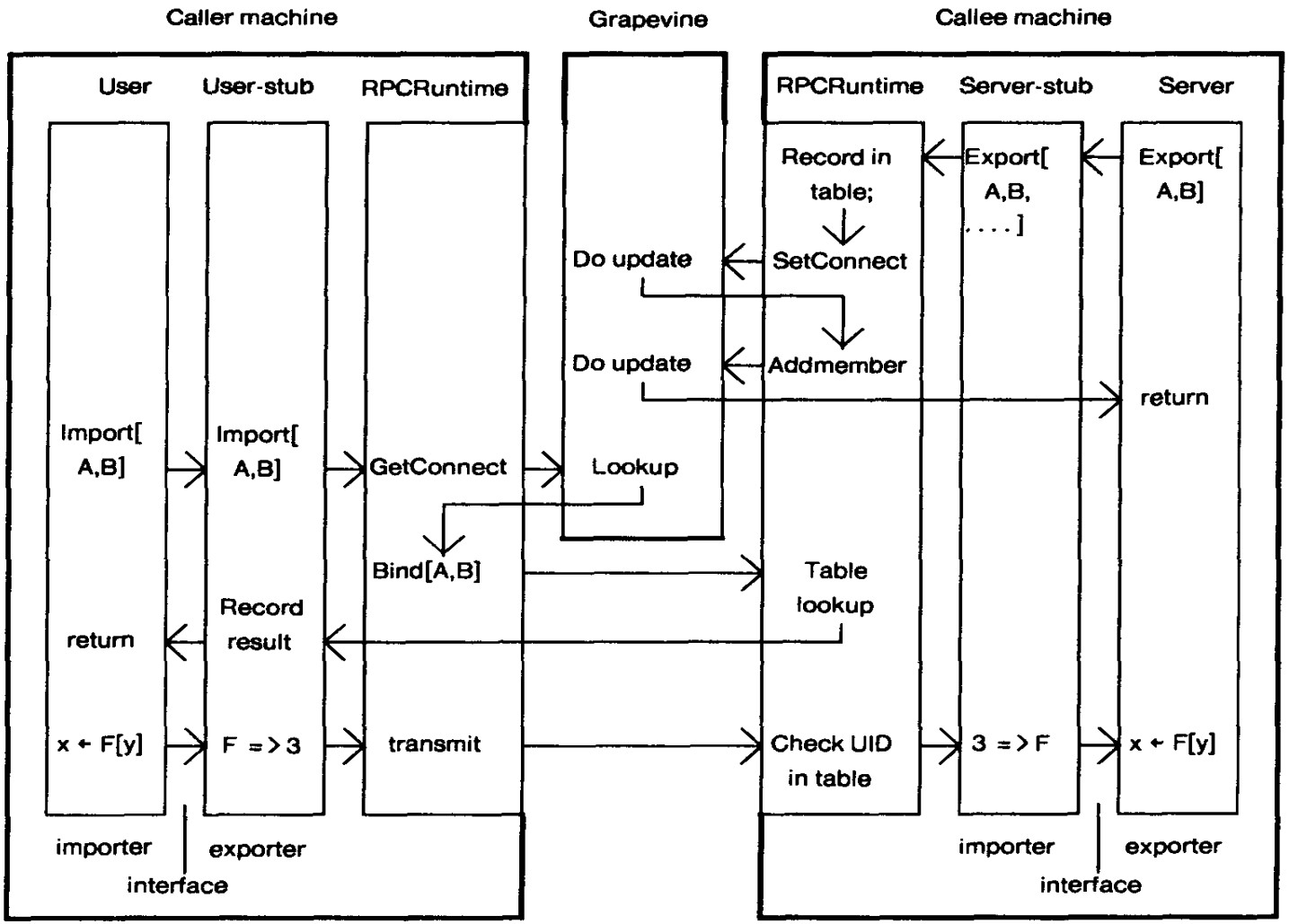
Interface

- Server uses ExportInterface(type, instance, procedure)
- Client uses ImportInterface(type, [instance])

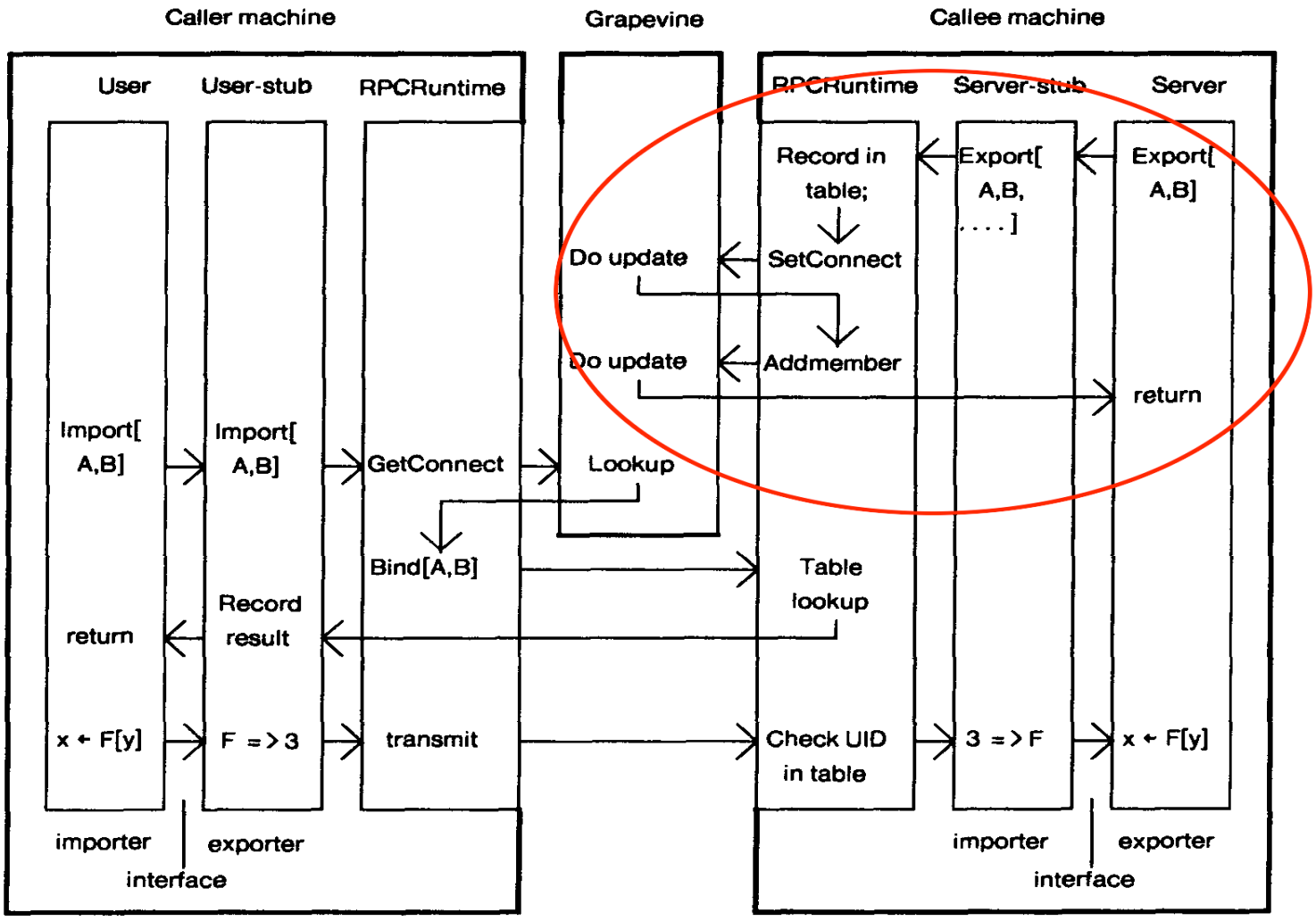
Look-up Table

- Unique binding identifier

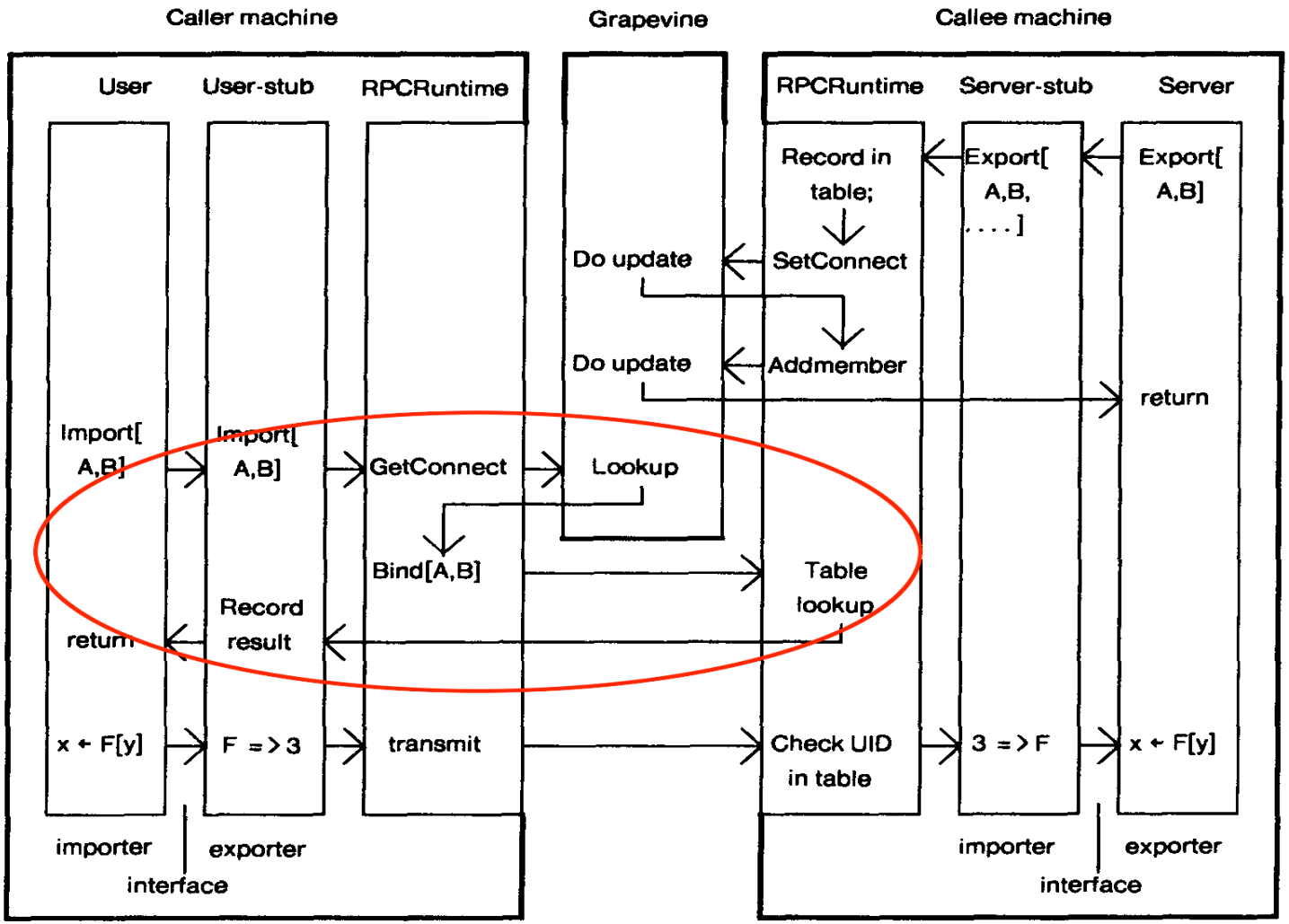
Binding Overview



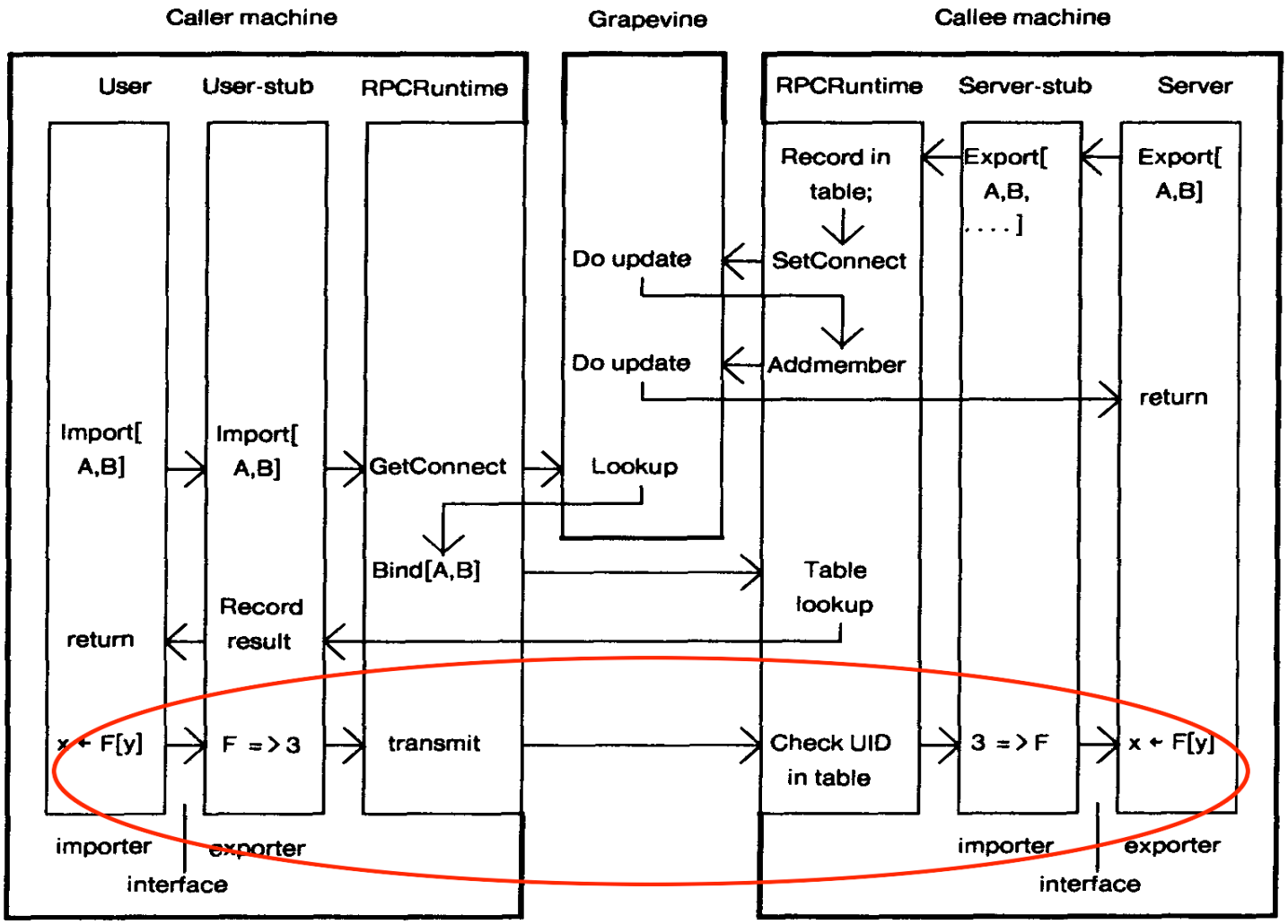
Binding: ExportInterface



Binding: ImportInterface



Binding


Caller machine
Grapevine
Callee machine
User
User-stub
RPCRuntime
RPCRuntime
Server-stub
Server
Record in table;
Export[A,B, ...]
Export[A,B]
Do update
SetConnect
Do update
Addmember
return
Import[A,B]
Import[A,B]
GetConnect
Lookup
Bind[A,B]
Table lookup
return
Record result
x ← F[y]
F => 3
transmit
Check UID in table
3 => F
x ← F[y]
importer
exporter
interface
importer
exporter
interface

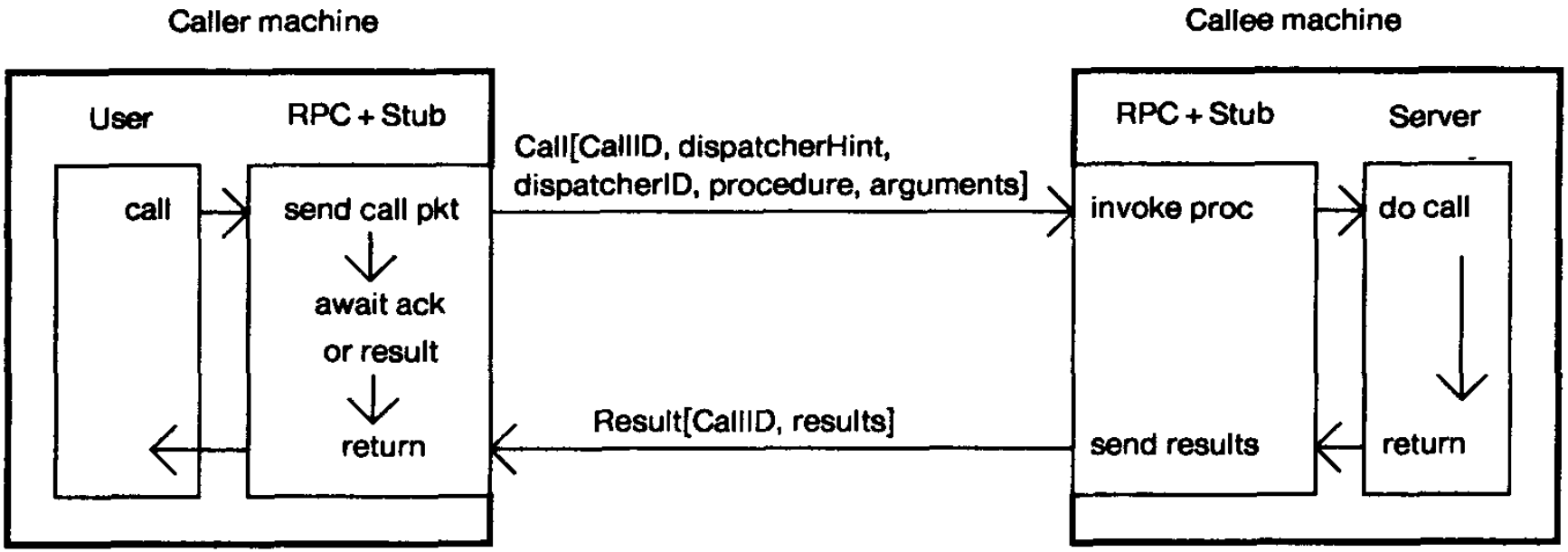
Packet Transport

- No specialized package-level protocol
  - Unsatisfactory experiments
- Small packets
  - Minimizing elapsed call time
  - No large data transfers
- One call, one return (or exception)

Transport Types

- Simple call: all arguments fit in one packet
- Complicated call: need to split into multiple packets

Simple Call



- Two packets
- Retransmission and Acknowledgement

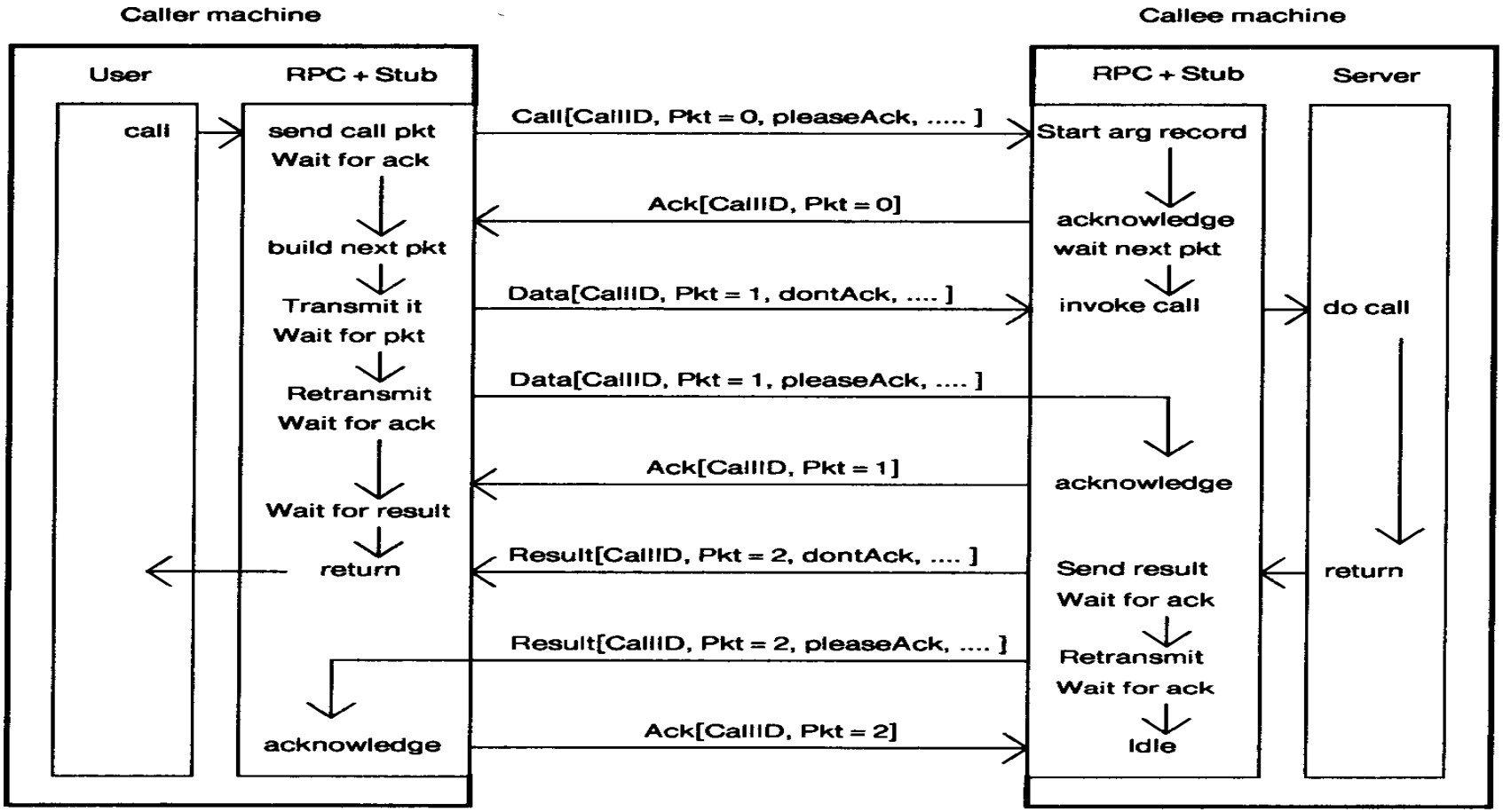
Call Details

- Call ID
  - Activity: one outstanding remote call
    - Machine ID
    - Process ID
  - Sequence Number: monotonic (global counter)

Look-up Table

- Unique binding identifier
- Call identifier

Complicated Call



- Probe packet
- Acknowledge all but the last packet

Exception Handling

- Signals
  - Dynamically scanning Mesa runtime system
- Exception packet
  - Handled by RPCRuntime

Optimizations

- Idle server processes
  - Process identifier swap
- Bypassing software layers
  - Modified network driver to treat RPC packets
  - RPC = Dominant
  - CHEATING

Performance

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

RPC Summary

- Advantages
  - Simple distributed interface for programmers
  - Portable (different stub generators)
  - Secure (future work)
- Disadvantages
  - Error handling: special network cases
  - Performance: two orders of magnitude slower than local procedure calls

ONC RPC (RFC 1831)

- Binding independent
  - Language interfaces
- Transport independent
  - Network protocols
- Authentication
- Asynchronous batching

RPC Conclusions

- Small code base (~2,200 lines)
- Distributed computing
- Bulk data transfer
- Security
  - Grapevine authentication
  - Packet data encryption

LRPC Outline

- Background
  - History
  - Environment
- Motivation and Goals
- Design
  - RPC problems
  - RPC optimizations
  - LPRC design
- Implementation
  - Binding
  - Calling

History

- B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy in 1990
- Exokernel in 1995
  - LPRC in ExOS based on Aegis's protected control transfer
  - More efficient than the Fastest RPC (259 µs vs 340 µs)
- Tornado in 2003
  - Protected Procedure Call (PPC)
    - Clustered Object call: client and server clustered objects
    - Stub Generator
    - Remote PPC: remote interrupts

Environment

- DEC SRC Firefly multiprocessor workstation
  - 5 MicroVAX II CPUs (1 MIPs each)
  - 16MB memory
- SRC Firefly RPC
  - Inferior performance to LRPC (464 μs vs 157μs for the simplest cross-domain call)
- Modula2+: strongly typed programming language, influenced by Mesa

Firefly RPC

- Close to Cedar RPC
- Grapevine is now a global call table
- Transport: UDP/IP
- Improvements
  - Direct thread wakeup from the Ethernet interrupt
  - Retaining packet buffer instead of UID
  - Same address space for packet buffer, Ethernet driver and interrupt handlers, sacrificing security
  - Special Ethernet operations in assembly language

LRPC Motivation

- RPC performance across domains is disappointing
- Most communication traffic are...
  - Cross-domain on the same machine
    - Cross-machine activity is very low on most systems
  - Simple, small values
    - Most procedure calls incur fewer than 50 bytes of parameters
- Independent threads exchanging large messages

LRPC Goals

- Performance, safety and transparency
  - Simple control transfer: execution within server domain
  - Simple data transfer: shared argument stack
  - Simple stubs: optimized
  - Concurrency: no locking

LRPC Design

- Problems in Cross-Domain RPC
- RPC Optimizations (for the above)
- LRPC = PPC + RPC

Problems in Cross-Domain RPC

- Stub overhead
- Message buffer overhead
- Access validation
- Message transfer
- Scheduling
- Context switch
- Dispatch

RPC Optimizations

- Shared buffer region
- Handoff scheduling
  - Direct context switch
- Passing arguments in register

LRPC = PPC + RPC

- PPC
  - Call to server procedure is a kernel trap
  - Kernel does validation and dispatches client thread to the server domain
- RPC
  - Similarity
    - Binding
    - Interfaces and stubs
  - Improvement
    - Calling

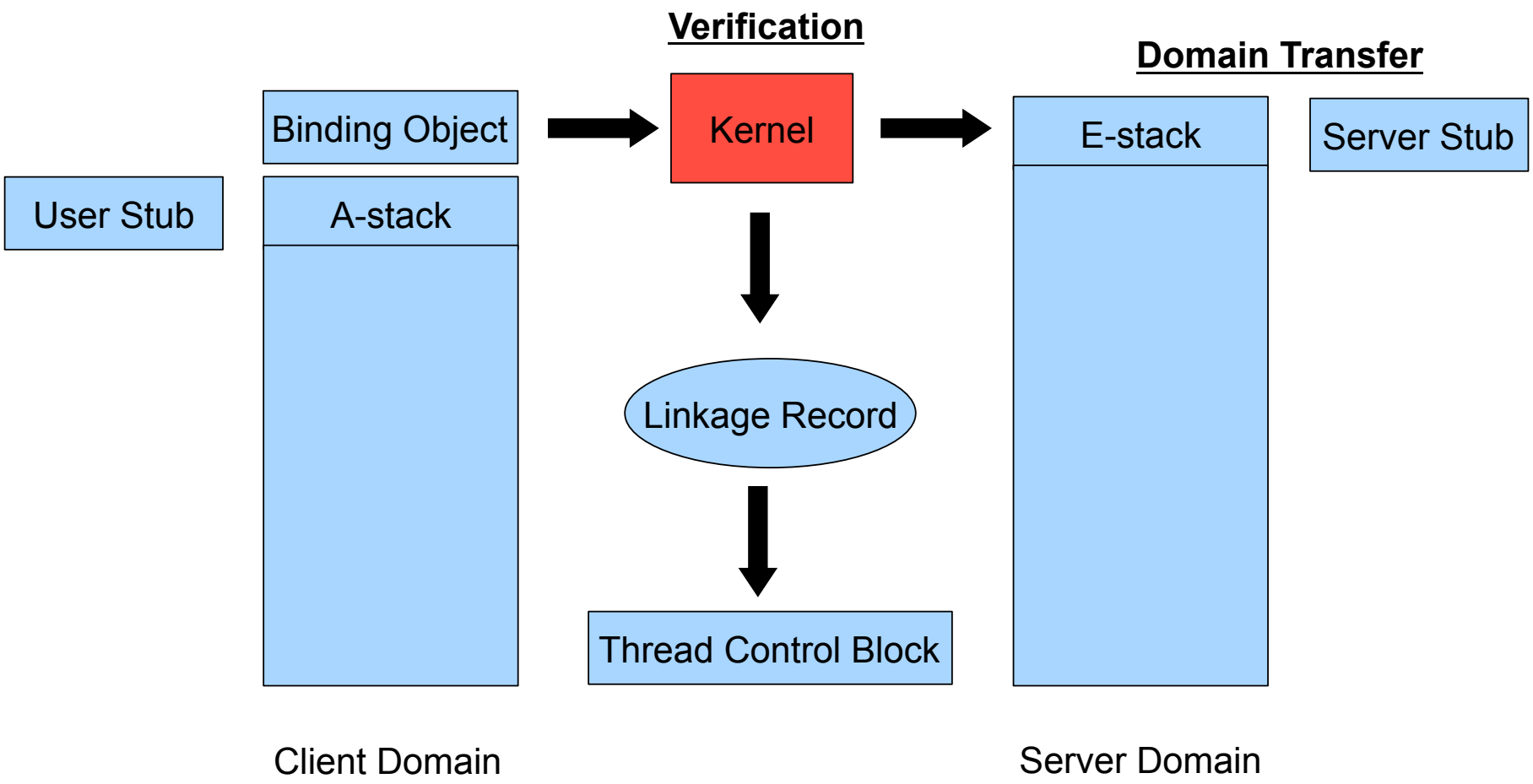
Binding

- Kernel
  - Validation: Grapevine
  - Linkage record: RPC's look-up table
- Clerk
  - Argument passing: RPCRuntime
  - Procedure descriptor list (PDL)
  - Argument stack (A-stack): mapped read-write and shared by both domains
- Binding Object: unique identifier

Interfaces and Stubs

- Interfaces written in Modula2+
- Stub generation in simple assembly language
- Portability

Calling


Verification
Domain Transfer
Binding Object
Kernel
E-stack
Server Stub
User Stub
A-stack
Linkage Record
Thread Control Block
Client Domain
Server Domain

Calling Details

- User stub
  - Traps a new A-stack, Binding Object and procedure ID into kernel
- Verification
  - Binding and procedure ID, finds Procedure Descriptor (PD)
  - A-stack, finds linkage record
  - No other thread is using current A-stack/linkage record
- Linkage Record
  - Caller return address and current stack point
  - Stored in caller's thread control block

Multiprocessing

- Caching domain contexts on idle processors (similar idea to RPC)
- Reduces TLB misses
- Process tag

Other Considerations

- Checking cross-machine calls
- Dynamic A-stack resizing
- Exception handling
  - Termination any time
  - Revoking Binding Object
  - Asynchronous termination

Performance

Table IV. LRPC Performance of Four Tests (in microseconds)

Test	Description	LRPC/MP	LRPC	Taos
Null	The Null cross-domain call	125	157	464
Add	A procedure taking two 4-byte arguments and returning one 4-byte argument	130	164	480
BigIn	A procedure taking one 200-byte argument	173	192	539
BigInOut	A procedure taking and returning one 200-byte argument	219	227	636

LRPC Summary

- Advantages
  - Efficient cross-domain RPC
  - Safety using protection calls
- Disadvantages
  - Exception handling is more complicated than RPC (revoking Binding Object)
  - Heavy dependence on kernel verification (end-to-end)

LRPC Conclusions and Comparison

- LRPC improves performance over RPC on same-machine, cross-domain calls
- Sufficient evidence that most calls are same-machine in practice
- LRPC has better security
- RPC is still the general protocol for NFS and distributed applications

Thank you!