Remote procedure calls

Haibo Chen

Some slides from Jinyang Li and Karthika Kothapally

RPC abstraction

Everyone loves procedure calls

- Transfer control and data on local programs

RPC goal: make client/server communication look like procedure calls

Easy to write programs with

- Procedure calls are a well-understood model
- RPC hides details of passing data between nodes

RPC vs. alternatives

Alternatives:

- Sockets
- MPI
- Distributed shared memory (later classes)
- Map/Reduce, Dryad (later classes)

RPC is very popular in programming distributed systems

- XML RPC
- Java RMI
- Sun RPC

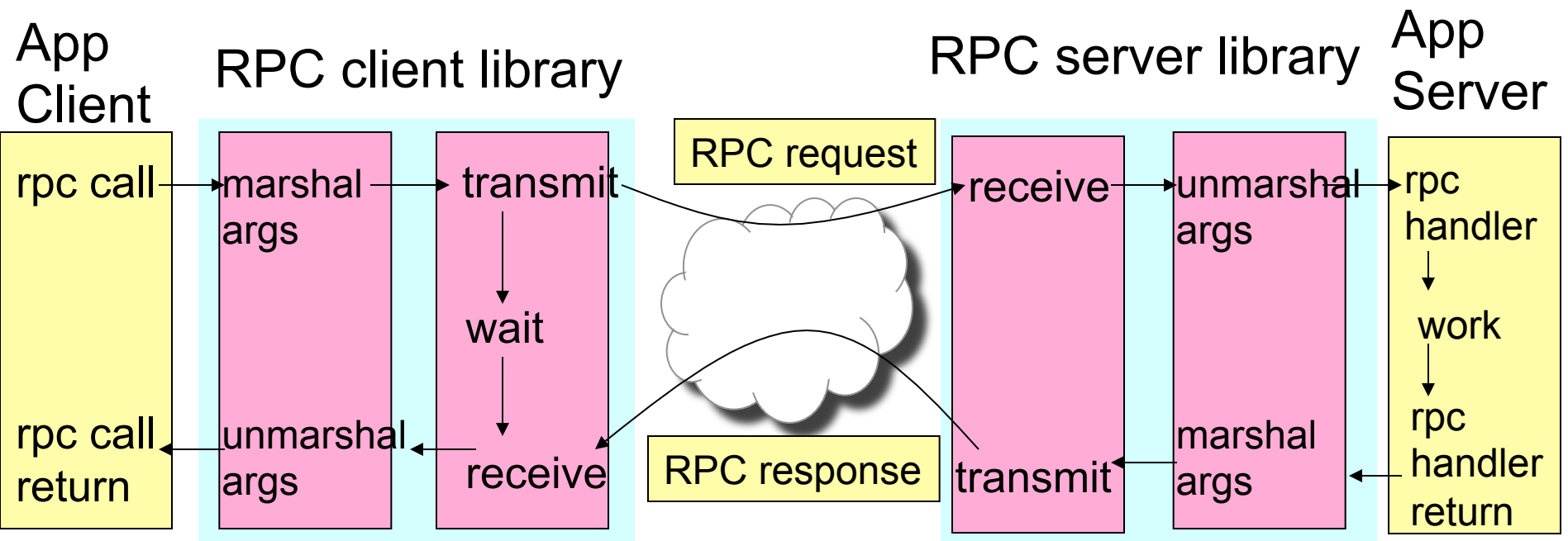
RPC architecture overview

Servers export their local procedure APIs

On client, RPC library generates RPC requests over network to server

On server, called procedure executes, result returned in RPC response to client

RPC architecture



Key challenges of RPC

- RPC semantics in the face of
  - Communication failures
    - delayed and lost messages
    - connection resets
    - expected packets never arrive
  - Machine failures
    - Server or client failures
    - Did server fail before or after processing the request?
  - Might be impossible to tell communication failures from machine failures

RPC failure semantics

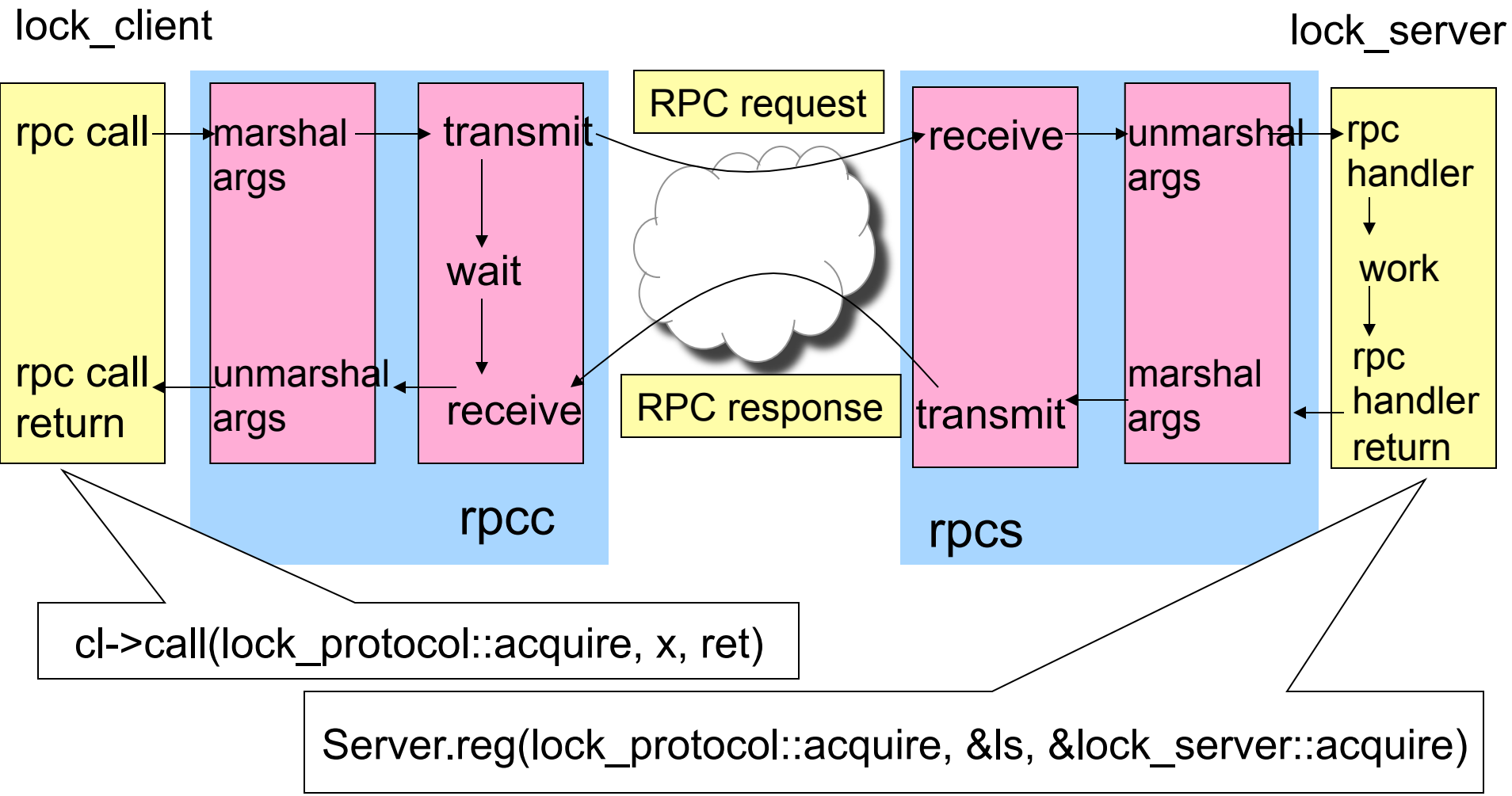
RPC might return “failure” instead of results

What are the possible outcomes in the face of failures?

- Procedure did not execute
- Procedure executed once
- Procedure executed many times
- Procedure partially executed

Desired semantics: at-most-once

YFS's RPC library



RPC semantics

Does yfs rpc implement at-most-once semantics?

How would the lack of at-most-once affect applications?

Interactions between threads and RPCs

Can a client hold locks across RPCs?

```
client_func()
{
   pthread_mutex_lock(&cl_lock);
   cl_rpcc->call(….)
   pthread_mutex_unlock(&cl_lock);
}
```

Should it do so?

Interactions between threads and RPC

How about this client side code?

```
client_func()
{
   pthread_lock(&cl_lock);
   for (vector<rpcc>::iterator i = list.begin(); i != list.end(); i++) {
      pthread_mutex_unlock(&cl_lock);
      (*i).call(….)
      pthread_mutex_lock(&cl_lock);
   }
}
```

Interactions between threads and RPCs

Can a server make a RPC call during RPC handler?

Lightweight Remote Procedure Call

Why LRPC?

Most communication traffic in operating systems is:

- Between domains on the same machine
- Simple rather than complex

In conventional RPC systems

- Local communication has been treated as an instance of remote communication
- Simple operations are considered in the same class as complex ones

Employing RPC technique for cross-domain communication would thus result in:

- Loss of performance
- Loss of structure

What is LRPC?

- A communication facility designed and optimized for communication between protection domains in the same machine
- Simplifies aspects of RPC
  - control transfer, data transfer, linkage, and stubs
- Used in small-kernel operating systems to avoid cost incurred by using RPC

LRPC Concepts

Execution model of LRPC is borrowed from "protected procedure call" of capability systems

Programming semantics and large-grained protection model of LRPC are borrowed from RPC

Four techniques of LRPC

- Simple control transfer
  - client's thread executes the requested procedure in server's domain
- Simple data transfer
  - Parameter-passing mechanism similar to procedure call
  - Shared argument stack eliminates redundant data copying
- Simple stubs
  - generation of highly-optimized stubs
- Design for concurrency
  - Avoids shared data structure bottle-necks
  - Benefits from speed-up of multiprocessor

Simple control transfer in LRPC

Client binds to a server interface before making first call

Call to server procedure by kernel trap

Kernel validates caller, creates a call linkage and dispatches client's thread directly to server domain

Client provides the server with an argument stack as well as its own thread of execution

On called procedure completion, control and results return through kernel back to the point of client's call

LRPC: Binding

Conceptually LRPC binding is similar to RPC binding but is different at lower level

At lower level:

Server provides Procedure Descriptor List, Used by kernel to allocate A-stacks and create linkage record

At completion, kernel returns to client a Binding object and A-stack list

LRPC: Some definitions

A-Stack:

- Arguments and return values
- Shared by client and server domains and mapped read-write

Linkage record:

- Records caller's return address
- One for each A-stack

Binding object:

- Client's key for accessing server's interface

LRPC: Calling

Client stub:

Client stub manages A-stacks as a LIFO

Takes an A-stack off the queue, push arguments onto it

Put A-stack address, Binding Object and procedure identifier into registers

Traps to kernel

LRPC Calling: Kernel

Verifies binding and procedure identifier, locates PD

Verifies A-stack and locates corresponding linkage

Find E-stack in server domain and update user thread stack pointer

Reload CPU virtual memory registers with those of the server domain

Performs an up-call into the server's stub at the address specified in the PD for the registered procedure

Simple data transfer

Argument copying:

- RPC requires data to be copied four times :
  - Stub to RPC message, client message to kernel, kernel to server and server to stack
- LRPC requires data to be copied only once:
  - From client stub's stack to shared A-stack from where server procedure can access
  - This optimization relies on a calling environment that uses a separate argument pointer

Procedures on same interface with A-stacks of similar size share A-stacks

Simple stubs

RPC makes the cross-domain and cross-machine calls transparent to lower level stubs

resulting in general and infrequently needed execution path.

LRPC:

Stubs blur boundaries between protocol layers to reduce the cost of crossing them

A simple LRPC needs only one formal procedure call (into client stub) and two returns (one out of server procedure and one out of client stub)

Stubs are generated at run-time from Modula2+ definition files

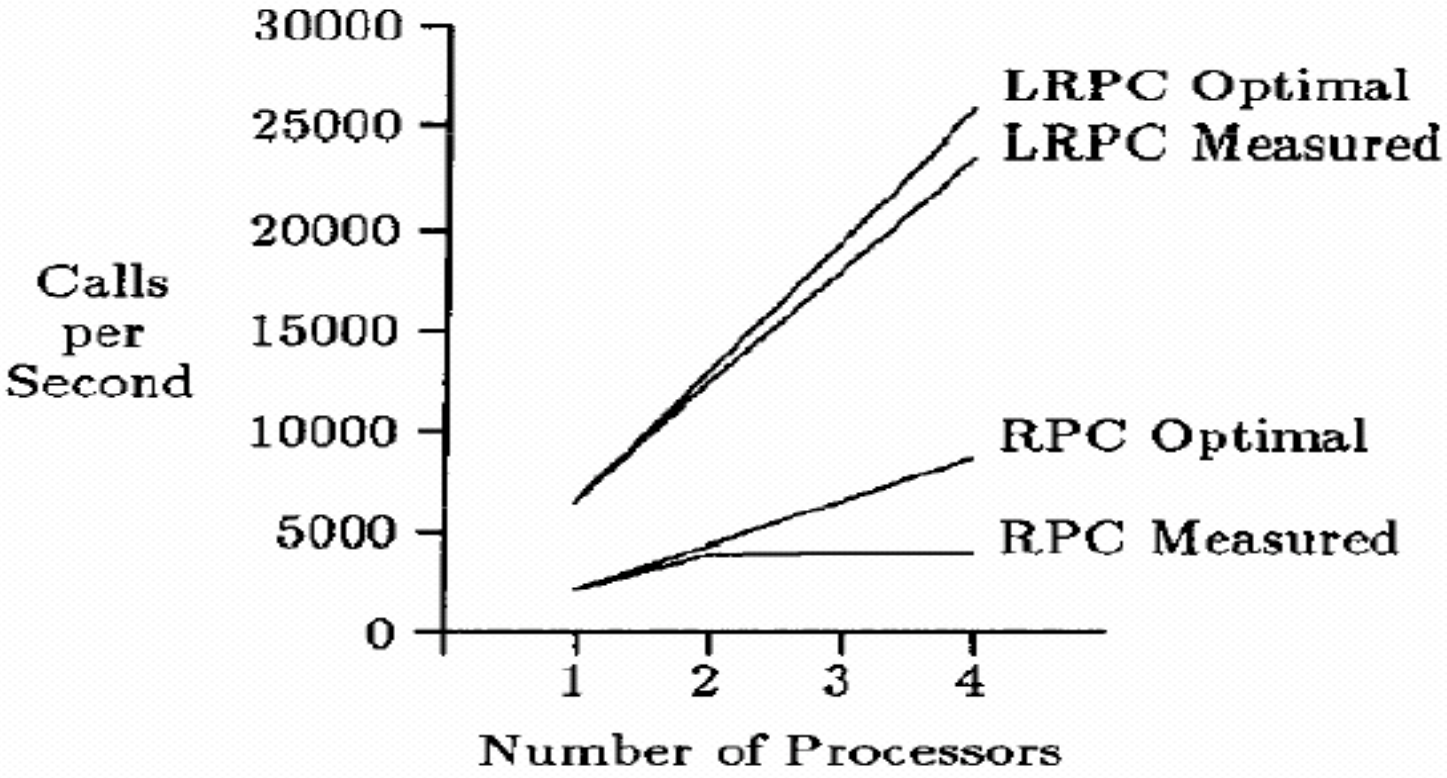
Complex parameters are handled by Modula2+ marshalling code

Design for concurrency

LRPC minimizes the use of shared data structures on critical domain transfer path

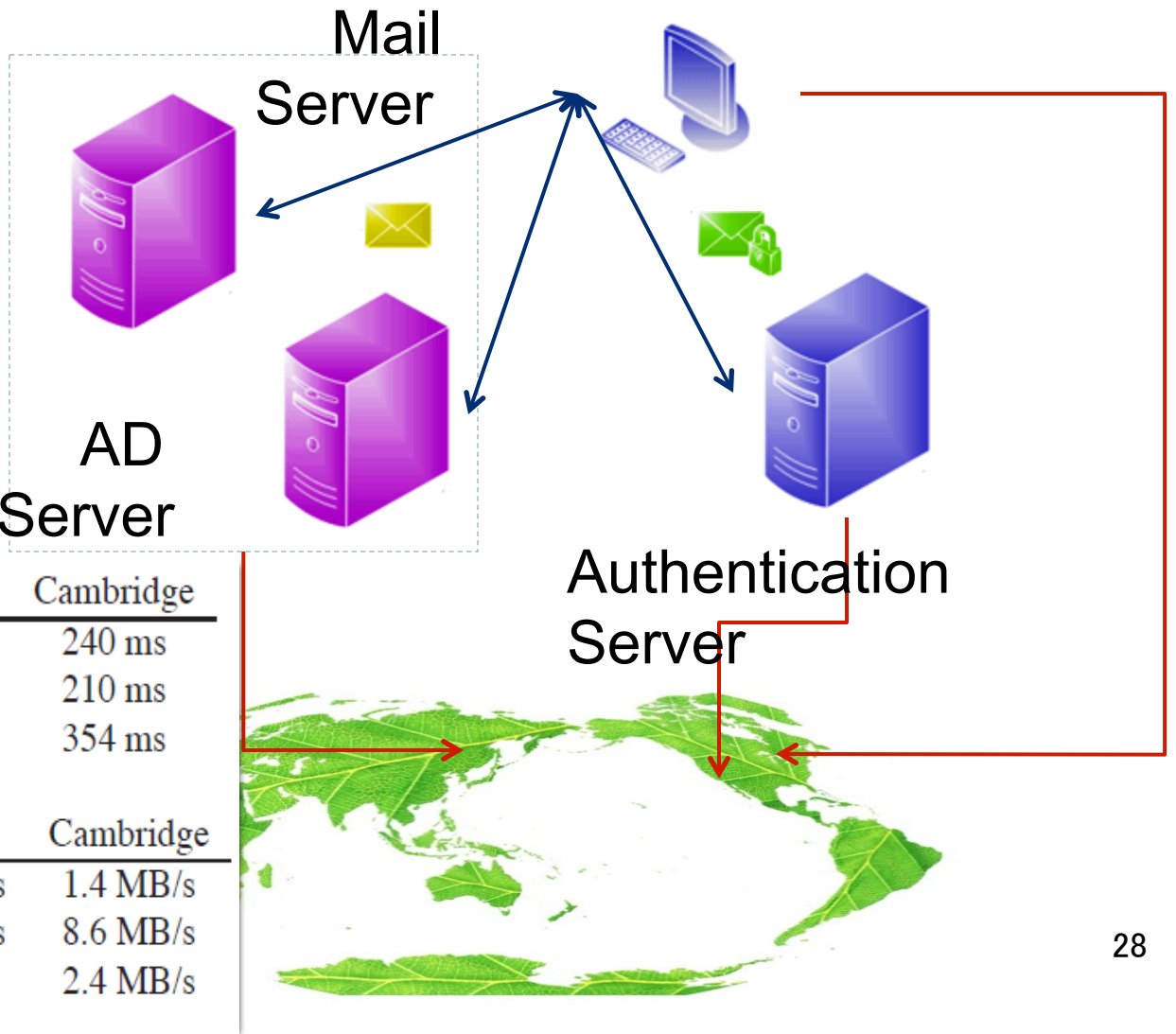
On shared-memory multiprocessors LRPC latency is reduced by caching domain contexts on idle processors

Performance of LRPC



RPC Chains: Efficient Client-Server Communication in Geodistributed Systems

Services are Geo-distributed

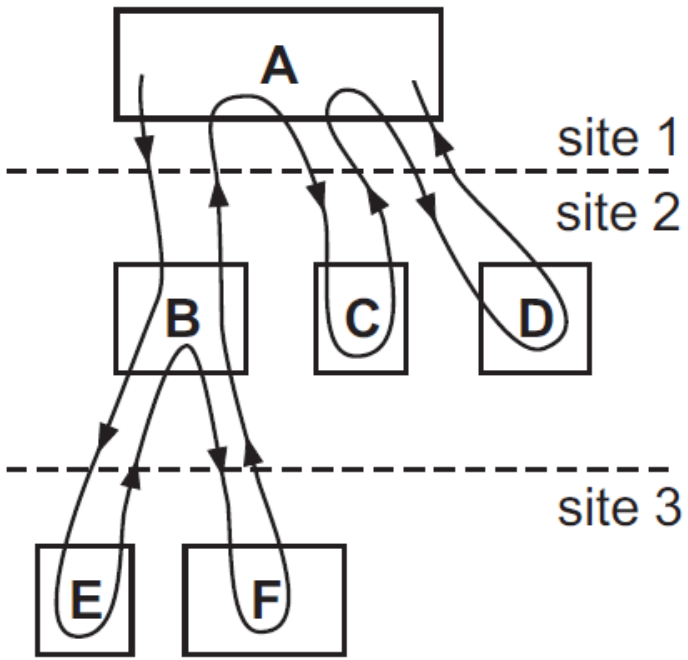


	Redmond	Beijing	Cambridge
Mt.View	32 ms	180 ms	240 ms
Redmond		146 ms	210 ms
Beijing			354 ms

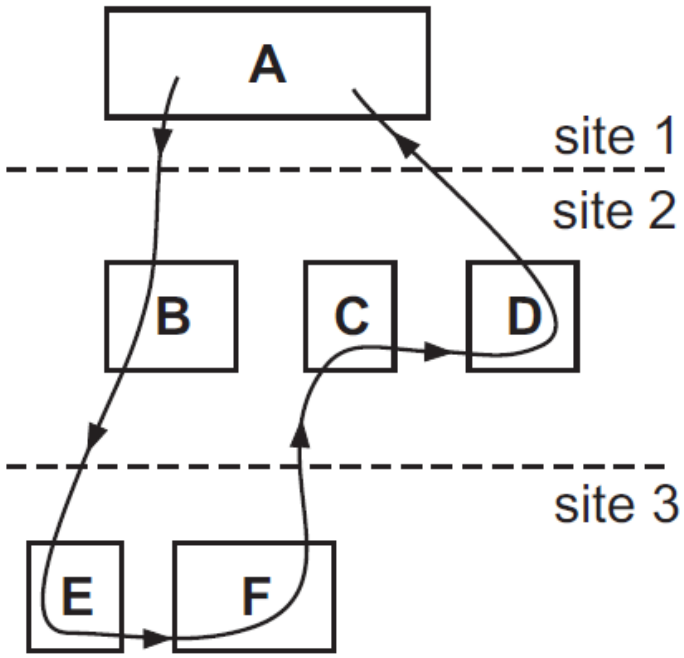
	Redmond	Beijing	Cambridge
Mt. View	6.3 MB/s	2.1 MB/s	1.4 MB/s
Redmond		8.5 MB/s	8.6 MB/s
Beijing			2.4 MB/s

What is RPC Chain

Standard RPC



RPC Chain



How would you implement this?

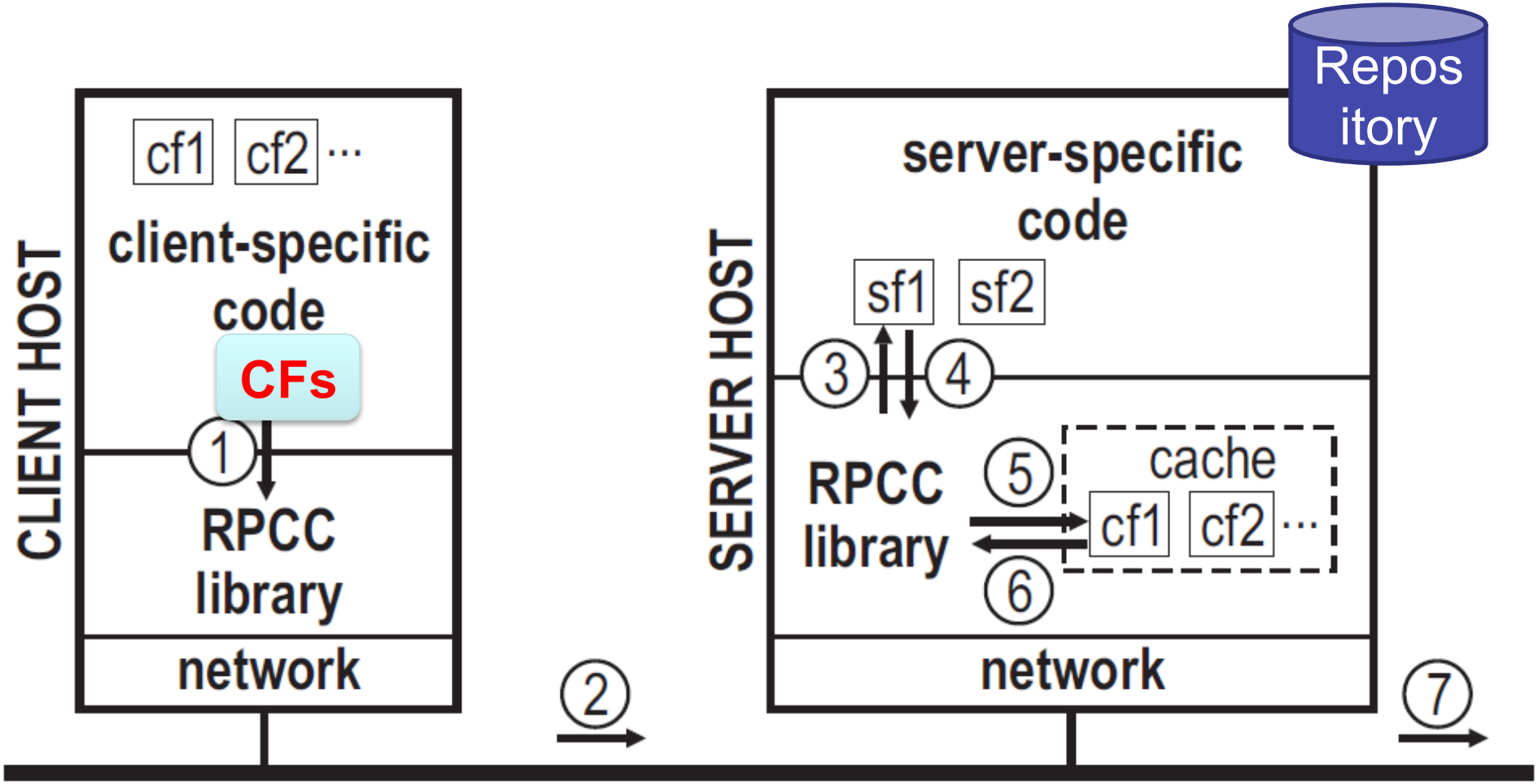
A simple but sometimes naive way:

- Client provide a static servers/service functions list

But…

- Dynamic chain
- Output reformat

Main mechanism



Insight into Chaining Function

```
// service function
object sf(object parmlist)
  // parmlist:  parameter list

// chaining function
nexthop cf(object state, object result)
  // state:  from client or earlier parts of chain
  // result:  from last preceding service function
  // returns next chain hop:
  //         (server, sf_name, parmlist,
  //         cf_name, state)
```

① call sf()

② call cf()

Reformat / chose hop

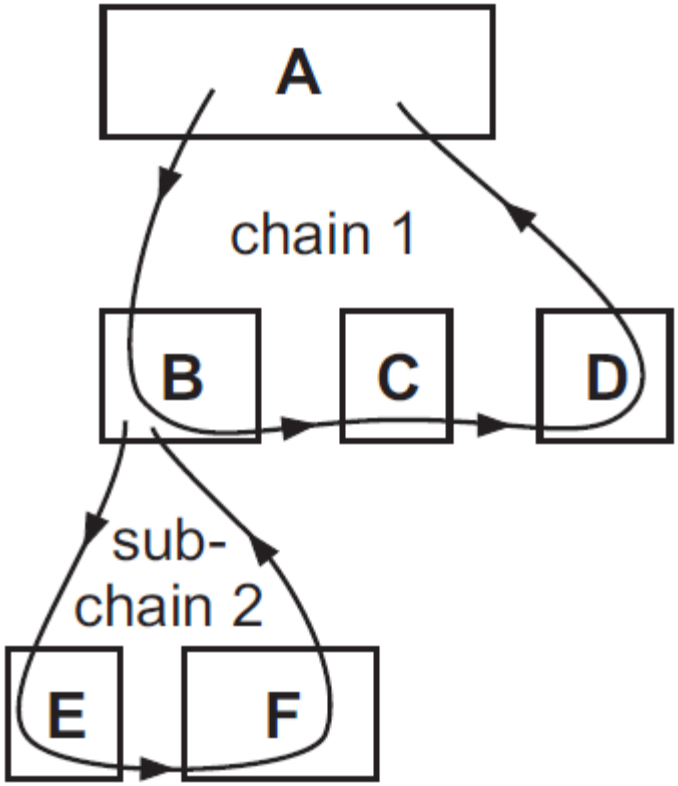
③ return strat_chain

```
chain_id start_chain(machine_t server,
        string sf_name, object parmlist,
        string cf_name, object state)
```

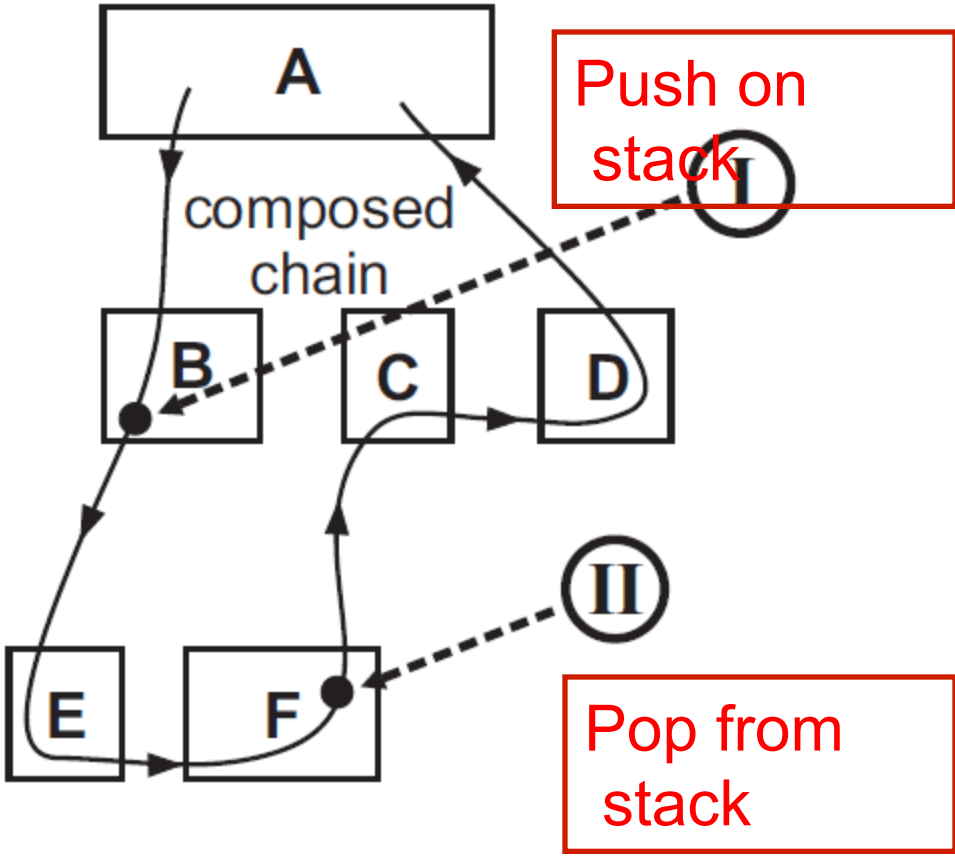
Figure 2: **(Top)** Signature of a service function (sf) and chaining function (cf). **(Bottom)** Signature of function that launches an RPC chain.

Nested and Composition chains

Nested



Composition



Fault Tolerance

Soft Exception

- Propagate upstream

Broken chains

- Detection - Counting timer
- Recovery - Chain-id and Cached results

Debugging and profiling

Virtual console

- Conveyed along the RPC chain

Interactive mode

- Execute chaining functions at the client
- Like standard RPC calls

Examples and Evaluation

Storage applications

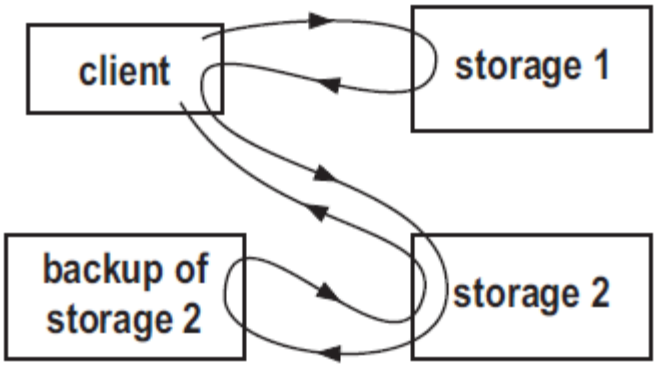
Web mail application

Micro-benchmarks

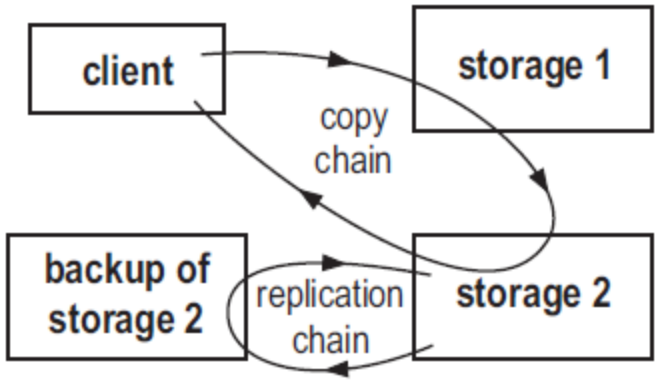
Application 1: Storage

Copy data from S1 to S2 with a backup server

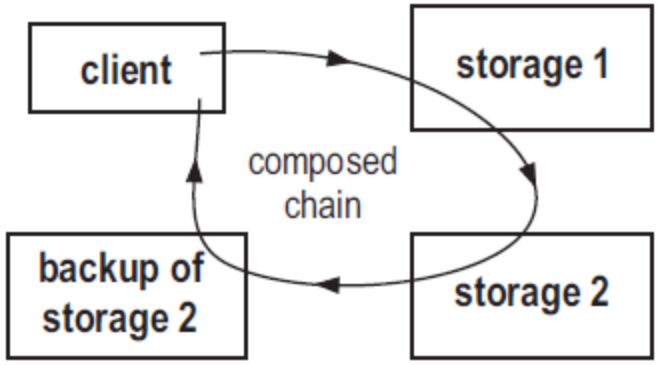
(a)



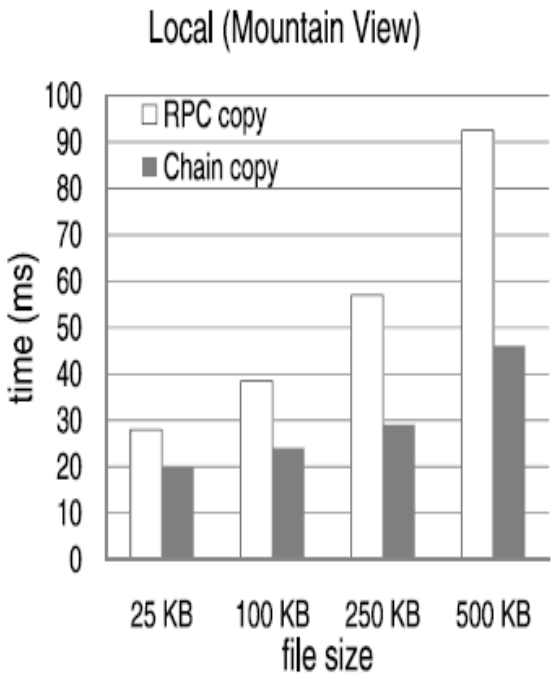
(b)

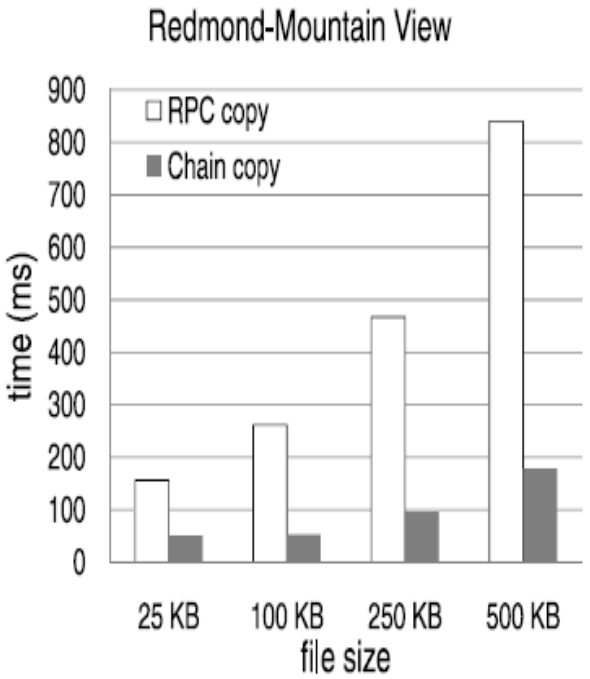


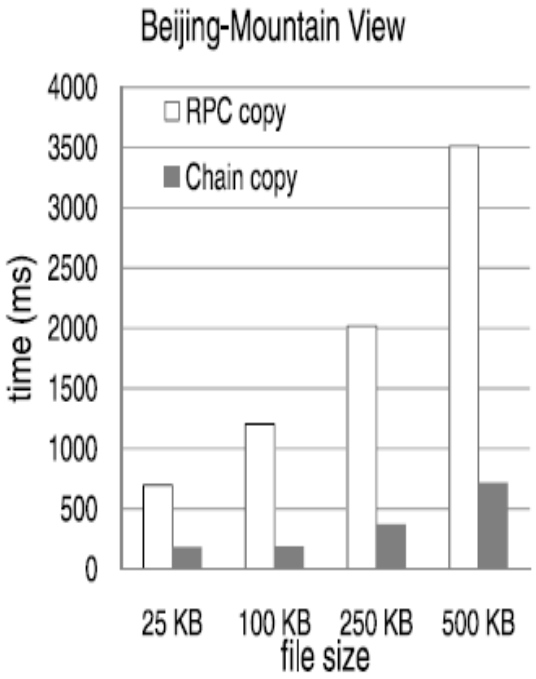
(c)



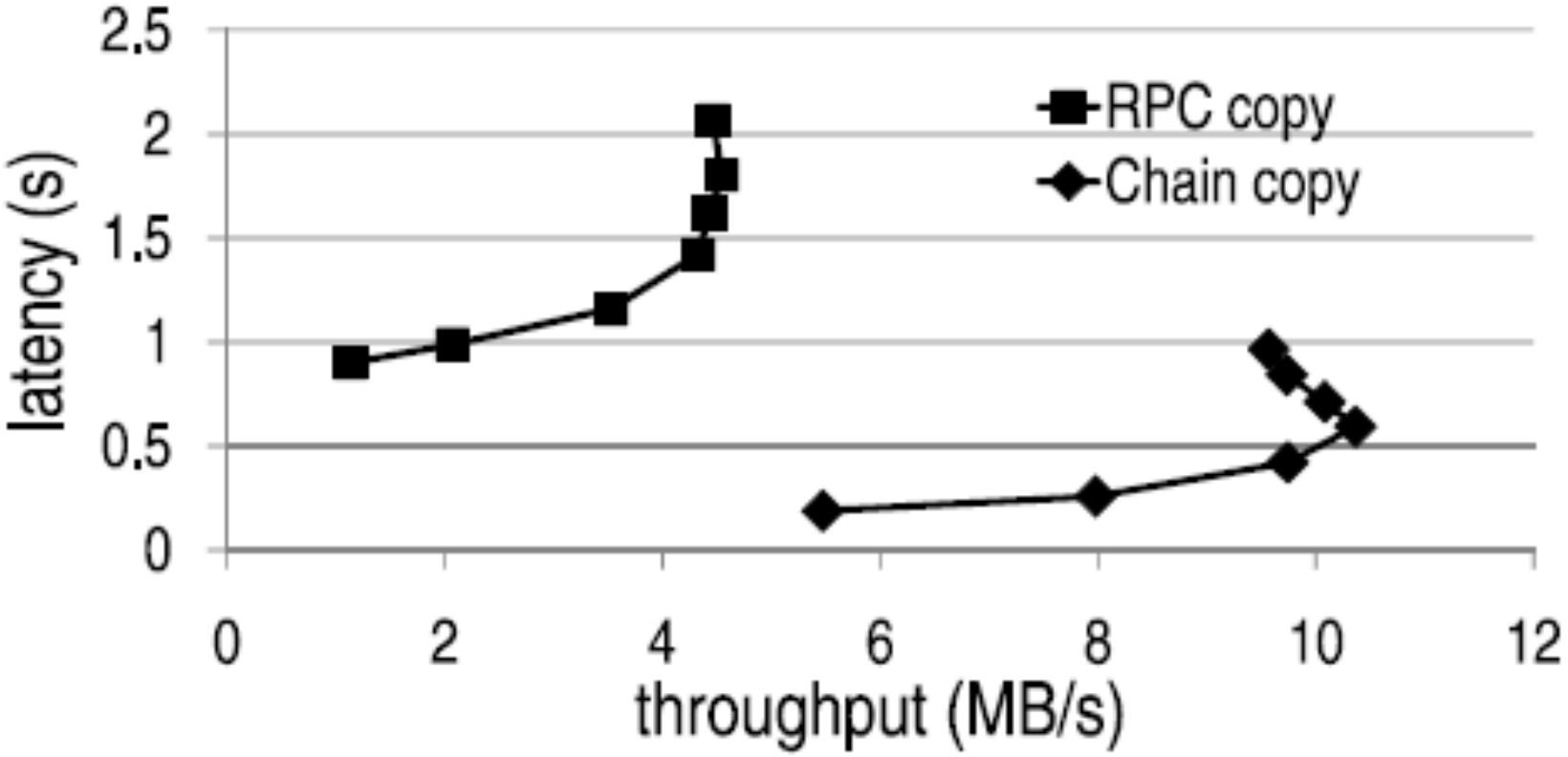
Copy Performance



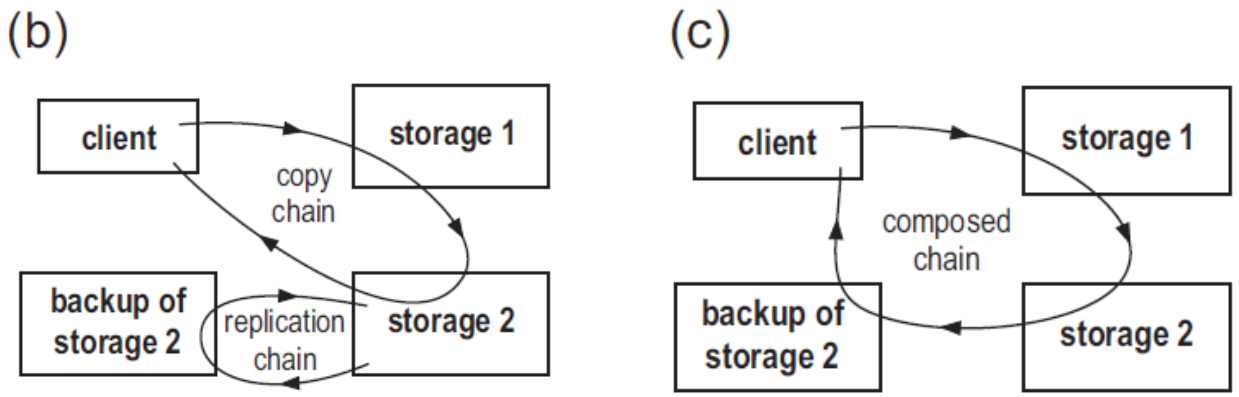


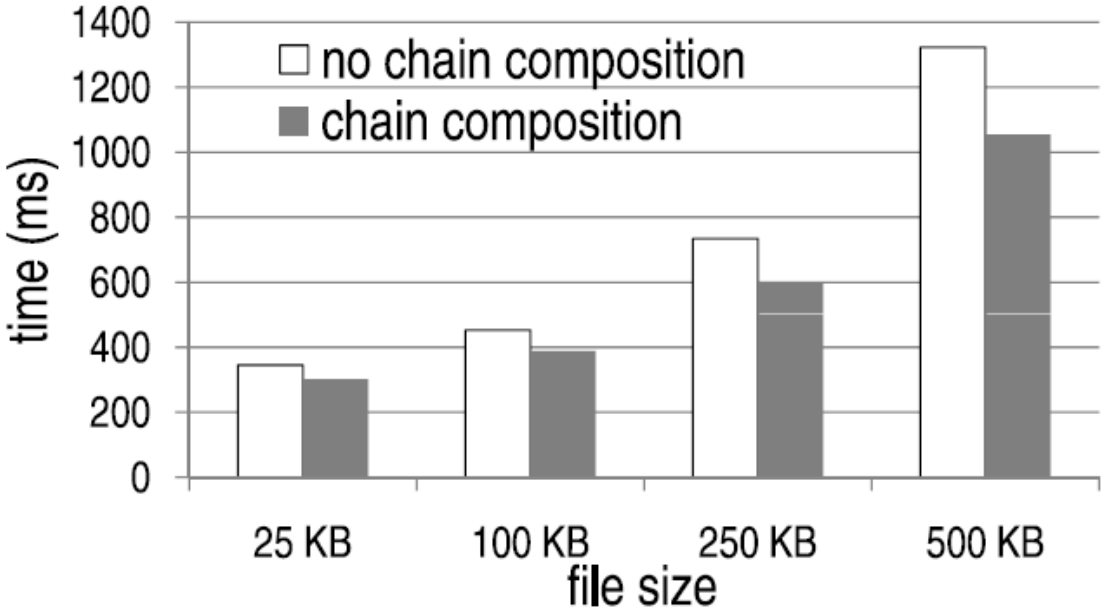


Copy Performance

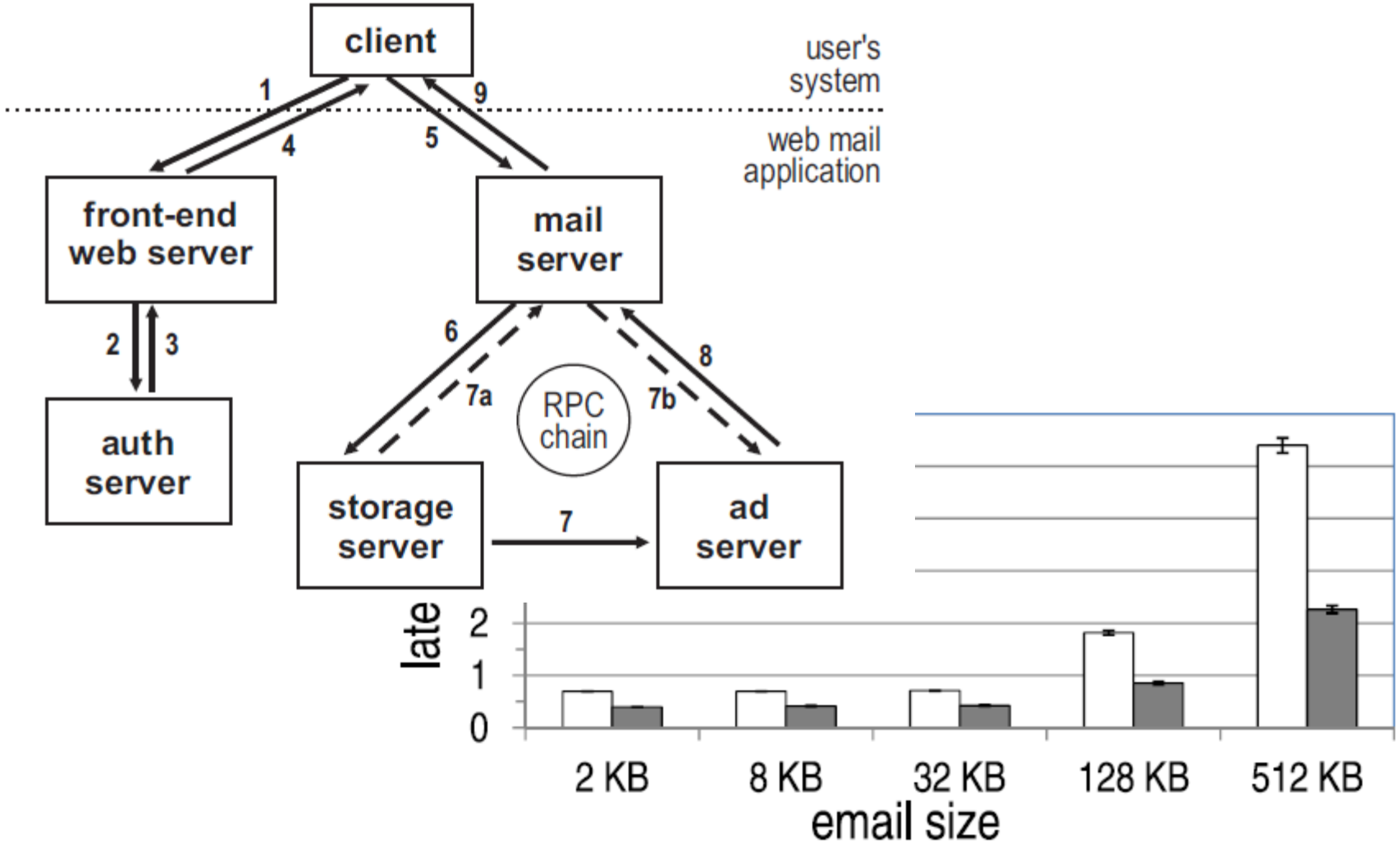


Benefit of Chain Composition





Application 2: Web Mail



Micro-benchmarks

- Overhead of chaining function(compile
-

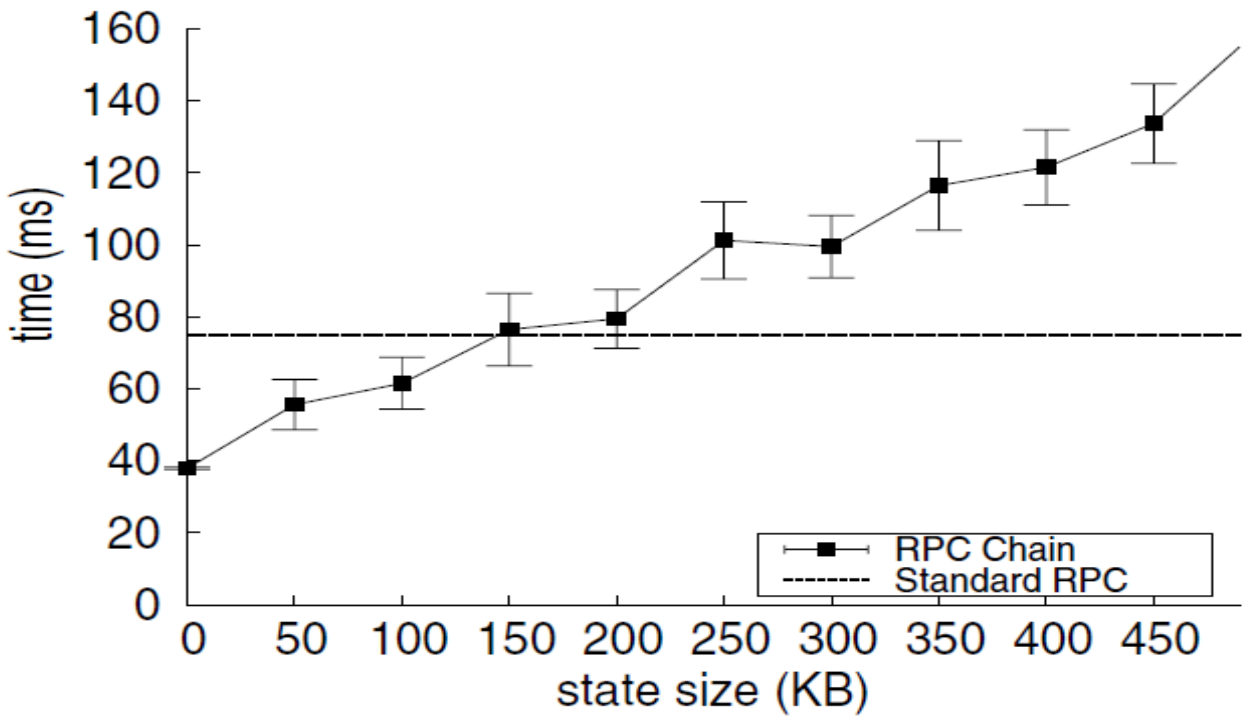


Figure 10: Execution time using an RPC chain versus standard RPC to call 2 servers.

DISCUSSION

Limitations

- Chaining state cannot always be sent (real time)
- Need to keep track of state (continuation)
- Terminating chains

Extensions

- Intermediate chain results
  - Return to client directly
- Dealing with large chaining states
  - Interactive mode
- Chaining proxy
  - Execute chaining functions

References

"Lightweight Remote Procedure Call" by B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, Proceedings of the 12th Symposium on Operating Systems Principles, pp. 102-113, December 1989

"Implementing Remote Procedure Calls" by A. D. Birrell and B. J. Nelson, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, February 1984

“rpc chains efficient client-server communication in geo distributed systems” by Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, Dahlia Malkhi. In Proceedings of NSDI, 2009.

Next Class

- Distributed Programming:
  - MapReduce, Piccolo