FIT3143

Parallel Computing

Week 3: Inter-Process Communications and RPC

Vishnu Monn and ABM Russel

Unit Topics

Week	Lecture Topic	Tutorial	Laboratory	Remarks
1	Introduction to Parallel Computing and Distributed Systems	None	Lab Week 01 - Introduction to Linux & Setting up VM (Not assessed)	Group formation for lab activities and assignments (Two students per group)
2	Parallel computing on shared memory (POSIX and OpenMP)	Tutorial Week 02 - Introduction to Parallel Computing	Lab Week 02 - C Primer (Not assessed)	
3	Inter Process Communications; Remote Procedure Calls	Tutorial Week 03 - Shared memory parallelism	Lab Week 03 - Threads (POSIX)	Assignment 1 specifications released
4	Parallel computing on distributed memory (MPI)	Tutorial Week 04 - Inter process communication	Lab Week 04 - Threads (OpenMP)	
5	Synchronization, MUTEX, Deadlocks	Tutorial Week 05 - Distributed memory parallelism	Lab Week 05 - Message passing interface	Assignment 2 specifications released
6	Election Algorithms, Distributed Transactions, Concurrency Control	Tutorial Week 06 - Synchronization	Lab Week 06 - Communication patterns	
7	Faults, Distributed Consensus, Security, Parallel Computing	Tutorial Week 07 - Transactions and concurrency	Lab Week 07 - Parallel data structure (I)	Assignment 1 due
8	Instruction Level Parallelism	Tutorial Week 08 - Consensus	1st assignment interview	
9	Vector Architecture	Tutorial Week 09 - Instruction level parallelism	Lab Week 09 - Parallel data structure (II)	
10	Data Parallel Architectures, SIMD Architectures	Tutorial Week 10 - Vector architecture	Lab Week 10 - Master slave	
11	Introduction to MIMD, Distributed Memory MIMD Architectures	Tutorial Week 11 - Data parallelism	Lab Week 11 - Performance tuning	Assignment 2 due
12	Recent development in Parallel Computing - Software, Hardware, Applications etc. (GPGPU etc.), Exam Revision	Tutorial Week 12 - Revision	2nd assignment code demo and interview	

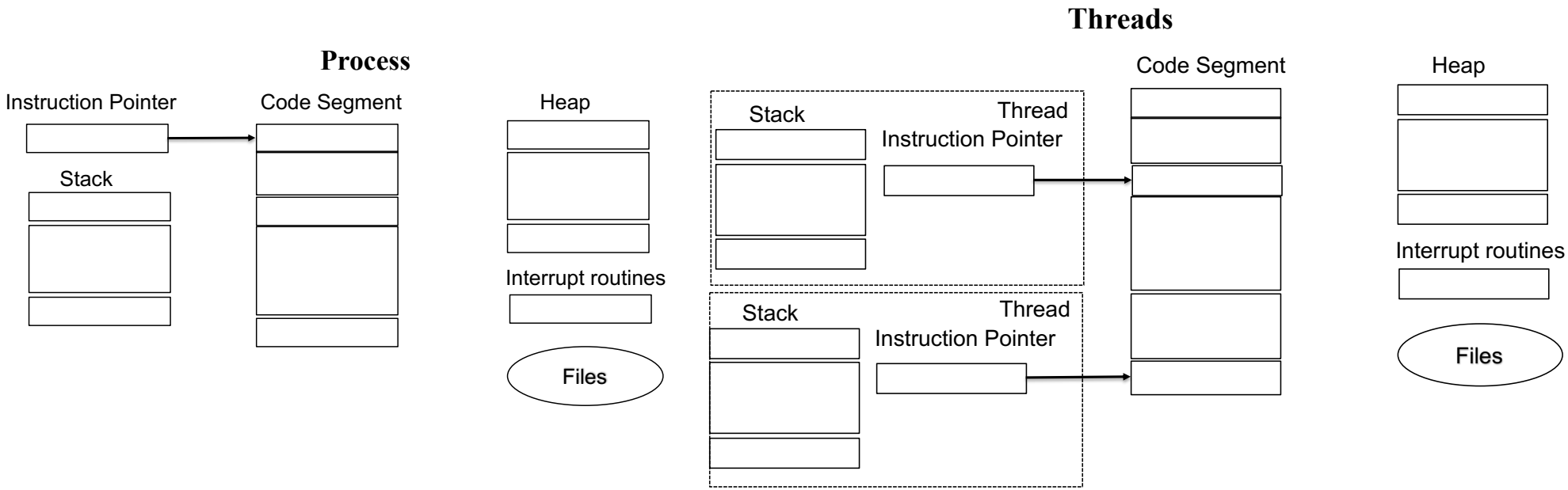
Assessment	Weight
Assignment 1	15%
Assignment 2	25%
Lab-work and Quizzes	10%
Final exam	50%

Today

- Recap: POSIX Threads and OpenMP
- Inter-Process Communications
- RPC

Recap

Process vs Threads



FLUX: Process vs Threads

Process vs Threads

JYGAFY

POSIX Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
  printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

Thread rank

```
void *threadFunc(void *pArg)
{
  int myNum = *((int*)pArg);
  printf( “Thread number %d\n”, myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  pthread_create(&tid[i], NULL, threadFunc, &tNum[i]);
}
```

OpenMP

Serial Code

```
#include <stdio.h>

int main() {
        int i;
        printf("Hello World\n");

        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

        printf("GoodBye World\n");
}
```

Parallel Code

```
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel
{
        int i;
        printf("Hello World\n");

#pragma omp for
        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

}
        printf("GoodBye World\n");
}
```

The Private & Shared Clause

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
  x = a[i]; y = b[i];
      c[i] = x + y;
    }
}
```

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Protect shared data

The “critical” pragma allows only one thread at a time to execute the update of sum.

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Reduction Clause

```
#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++) {
    sum += a[i] * b[i];
  }
```

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in "global" variable

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

Scheduling Clauses

schedule(static [,chunk])

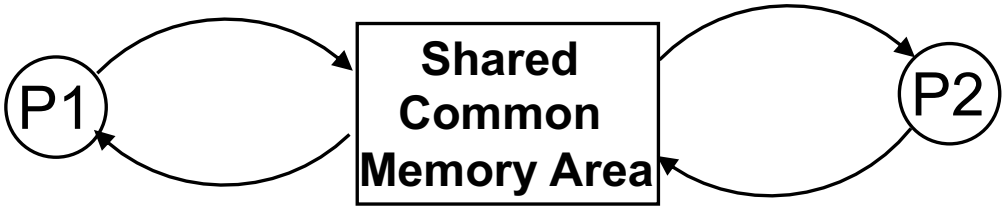
schedule(dynamic[,chunk])

schedule(guided[,chunk])

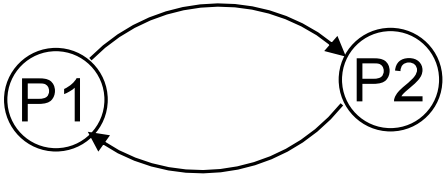
Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Inter-process Communication (IPC)

IPC: Information sharing



Shared data approach



Message passing approach

Message Passing System

- A *message-passing system* is a sub-system of a distributed system that provides a set of message-based IPC protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers.

A good message passing system

- Simplicity
- Uniform semantics
- Efficiency
- Reliability
- Correctness
- Flexibility
- Security
- Portability

A typical message structure

–Header

- Addresses
  - ✓Sender address
  - ✓Receiver address
- Sequence number
- Structural information
  - ✓Type
  - ✓Number of bytes

–Message

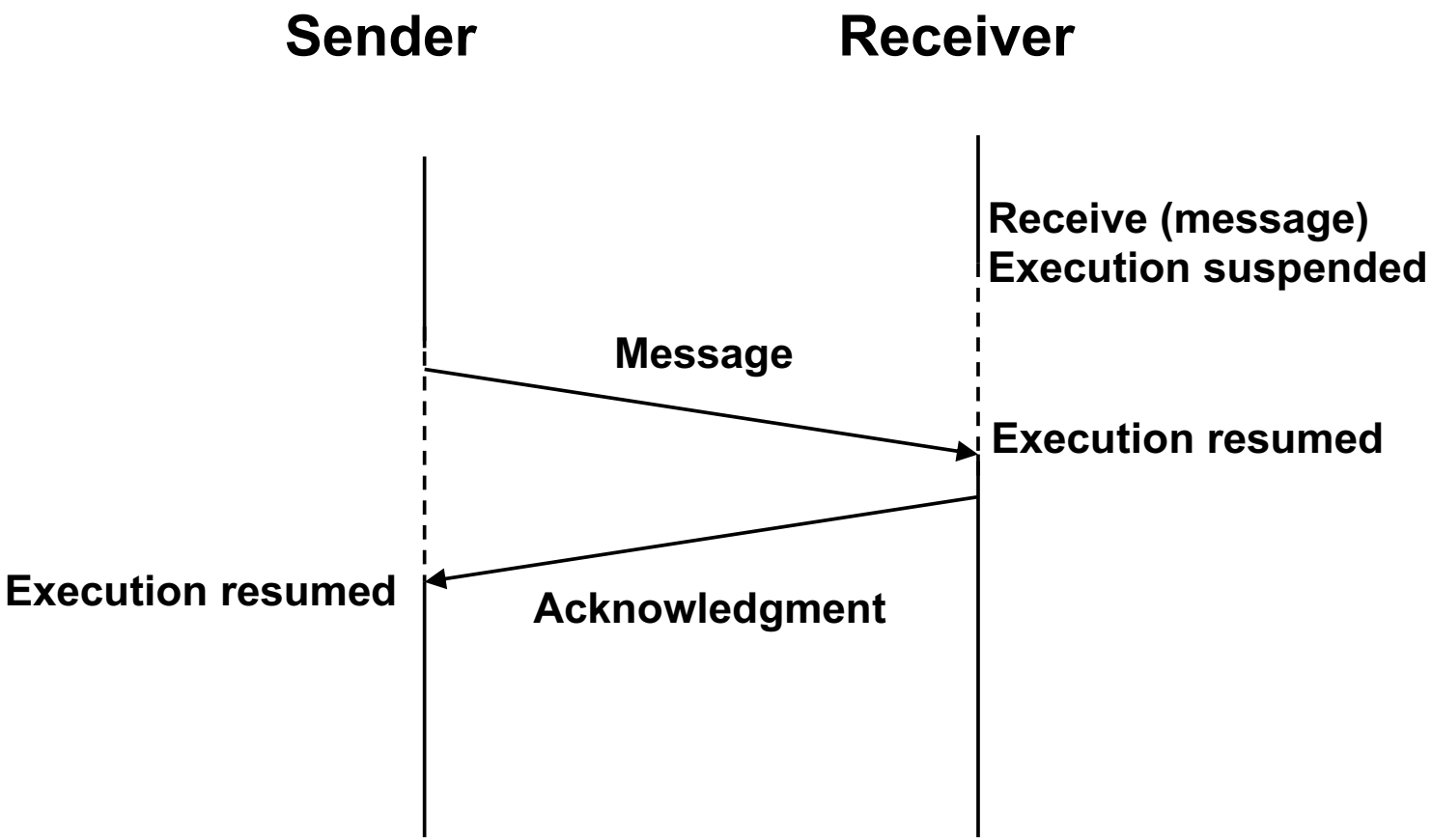
Issues

- Identity related
- Network Topology related
- Flow control related
- Error control and channel management

Synchronization in IPC

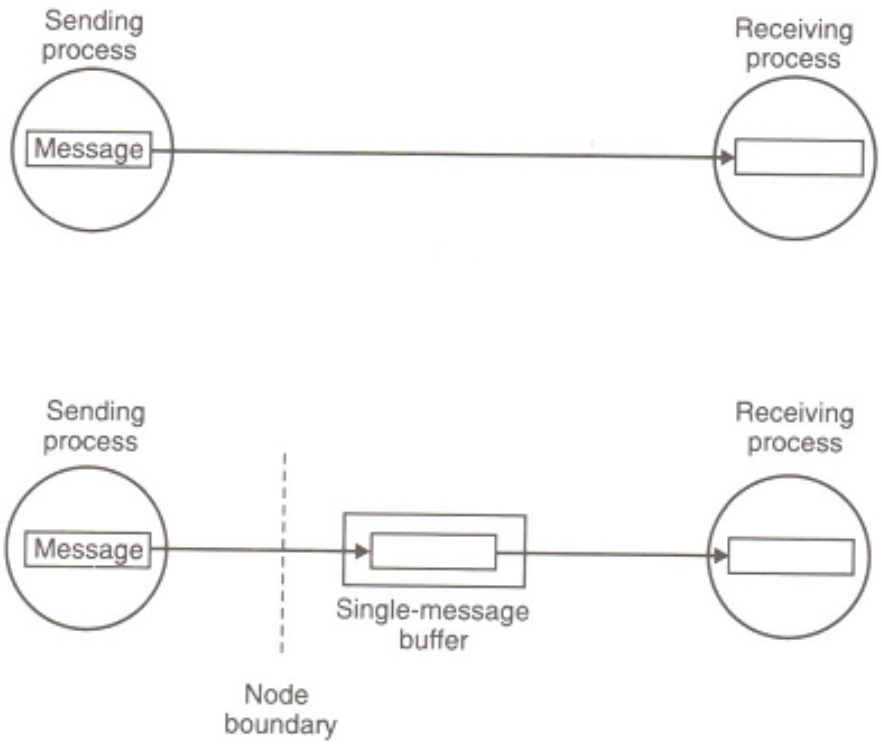
- Send primitive
  - Blocking
  - Non blocking
- Receive primitive
  - Blocking
  - Non Blocking
    - Polling
    - interrupt

Synchronous & Asynchronous Communication

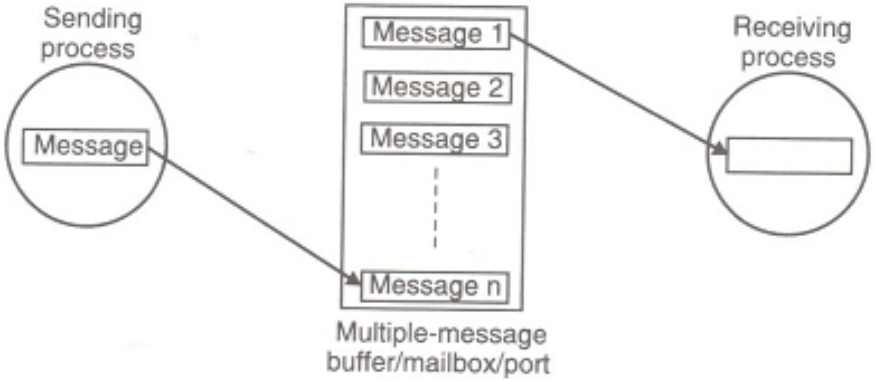


Synchronous mode with blocking send and receive primitives

Buffering



Synchronous System

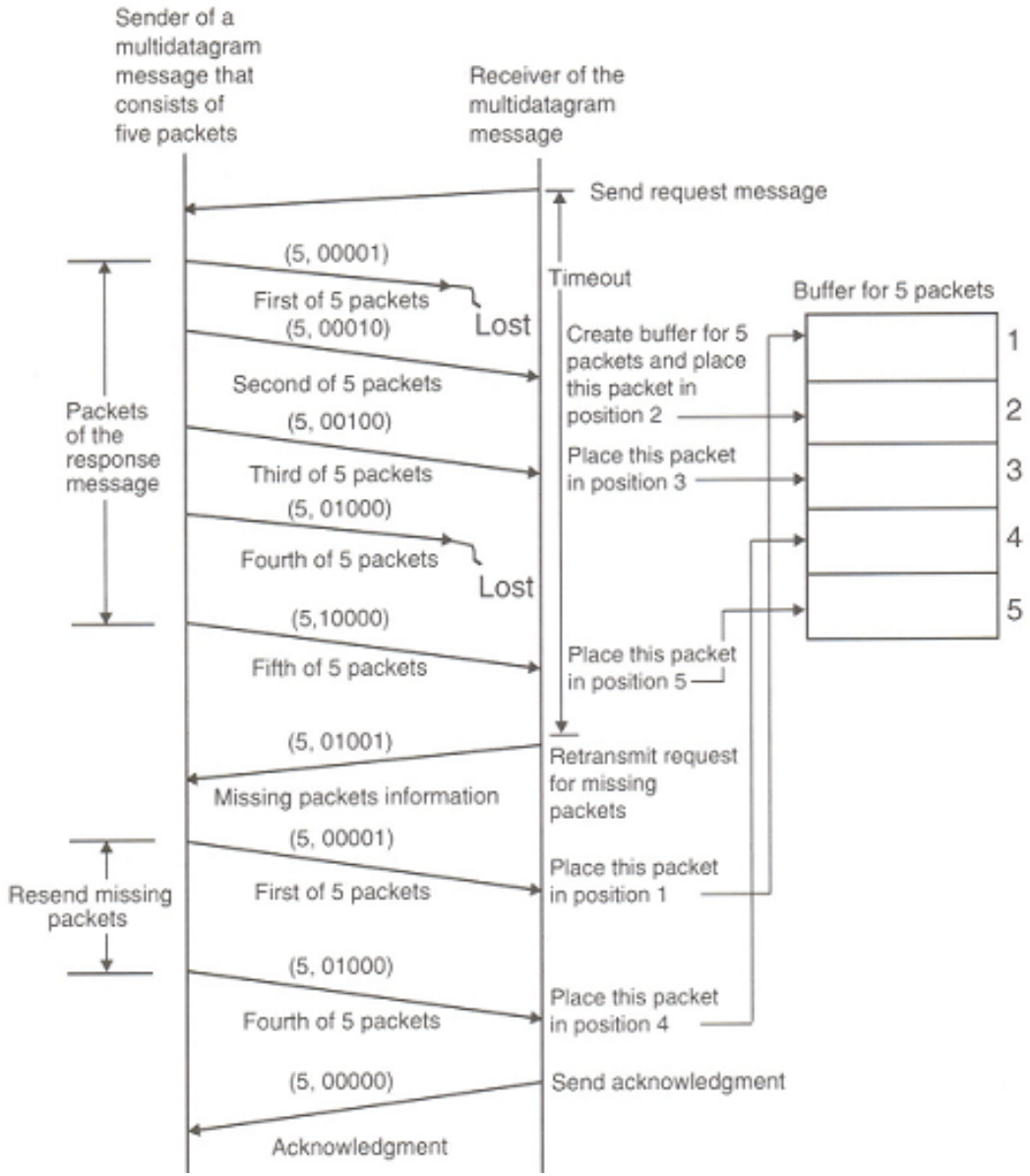


Asynchronous System

Multi-datagram Messages

- MTU (maximum transfer unit)
- Messages may be single-datagram messages or multi-datagram messages
- Assembling and dissembling is the responsibility of message passing system.

Using Bitmap for multidatagrams

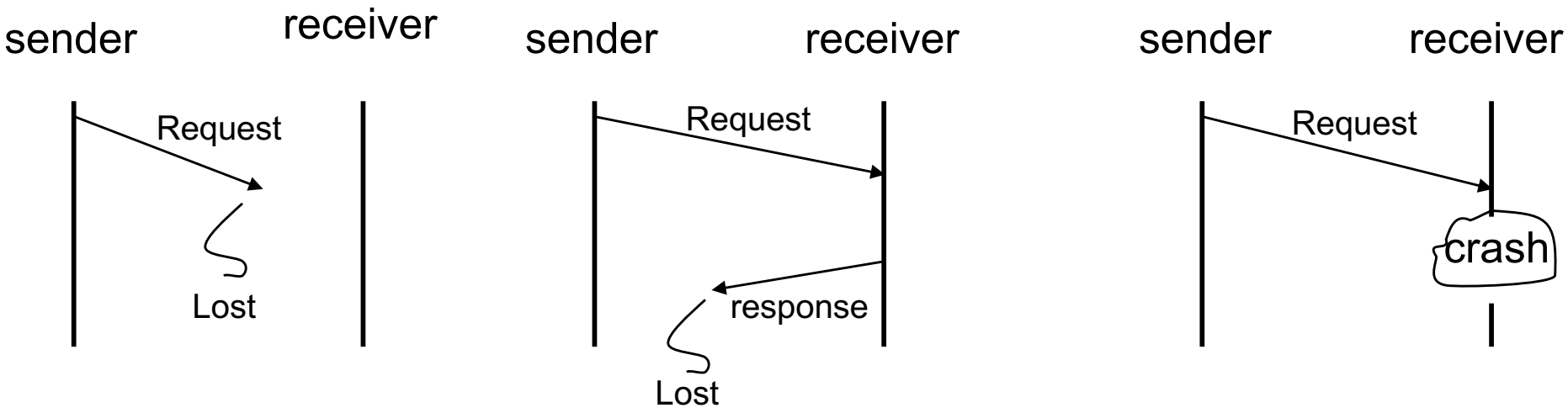


Encoding Decoding

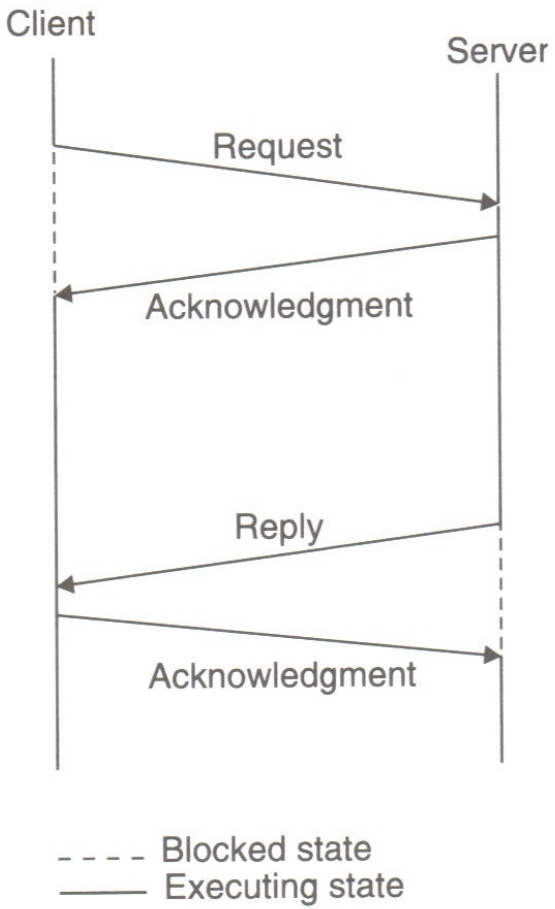
- Encoding/Decoding is needed if sender and receiver have different architecture
- Even for Homogeneous Encoding/Decoding is needed for using an absolute pointer and to know which object is stored in where and how much storage it requires

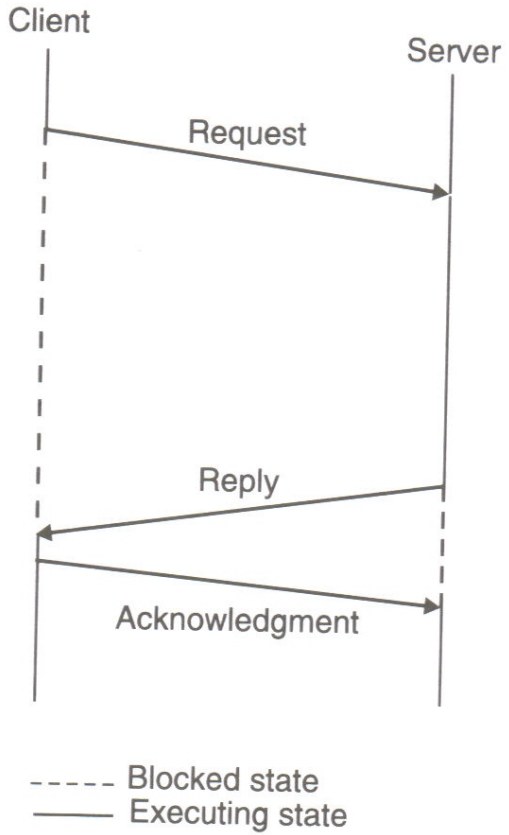
Failure handling

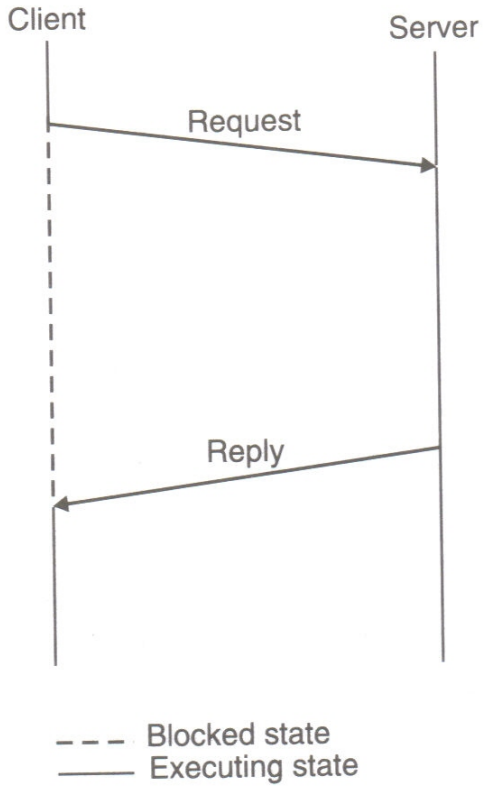
- Failure classification
  - Loss of request message
  - Loss of response message
  - Unsuccessful execution of the request



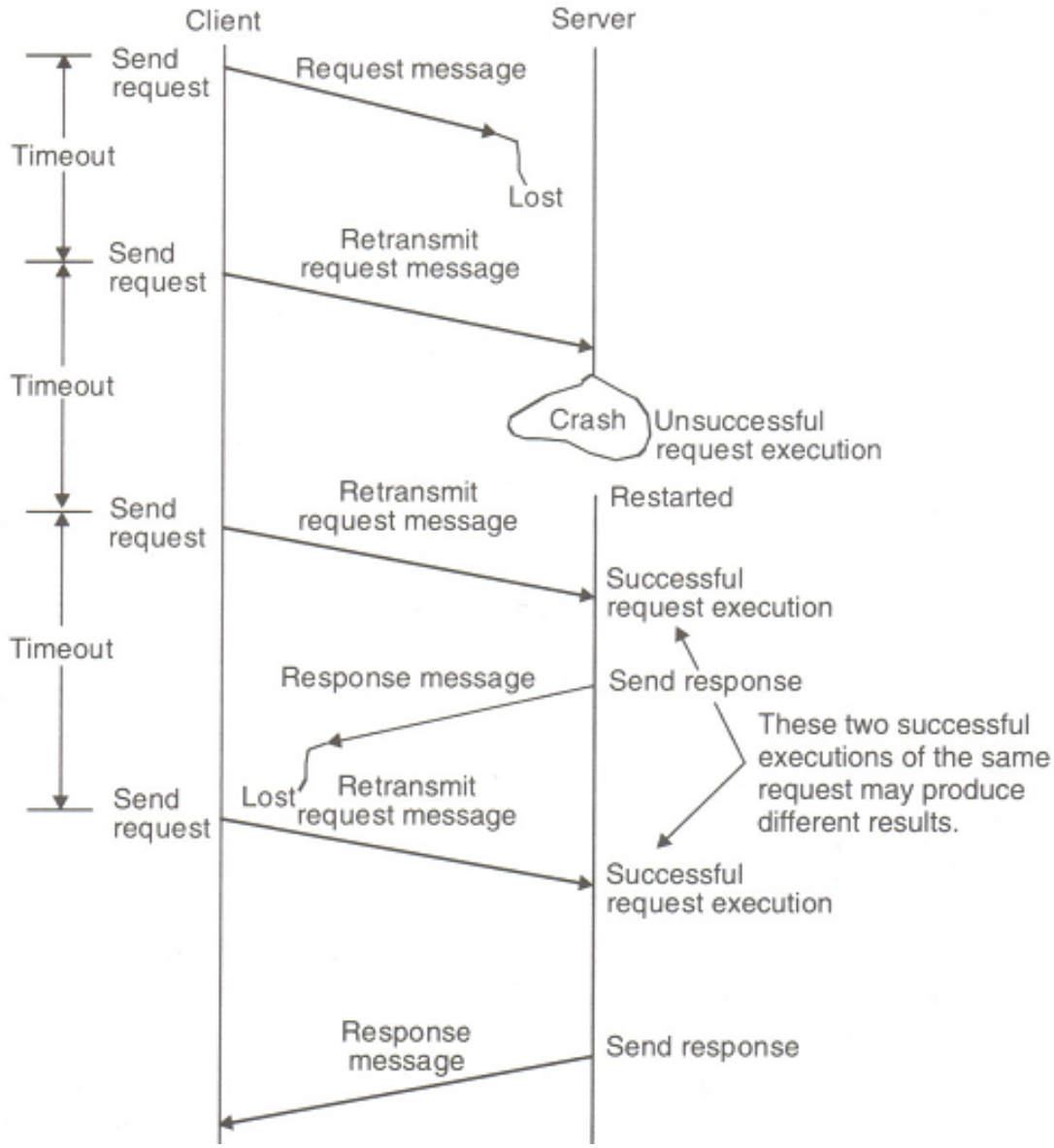
Blocking


Client
Server
Request
Acknowledgment
Reply
Acknowledgment
Blocked state
Executing state


Client
Server
Request
Reply
Acknowledgment
Blocked state
Executing state


Client
Server
Request
Reply
Blocked state
Executing state

Fault Tolerant System



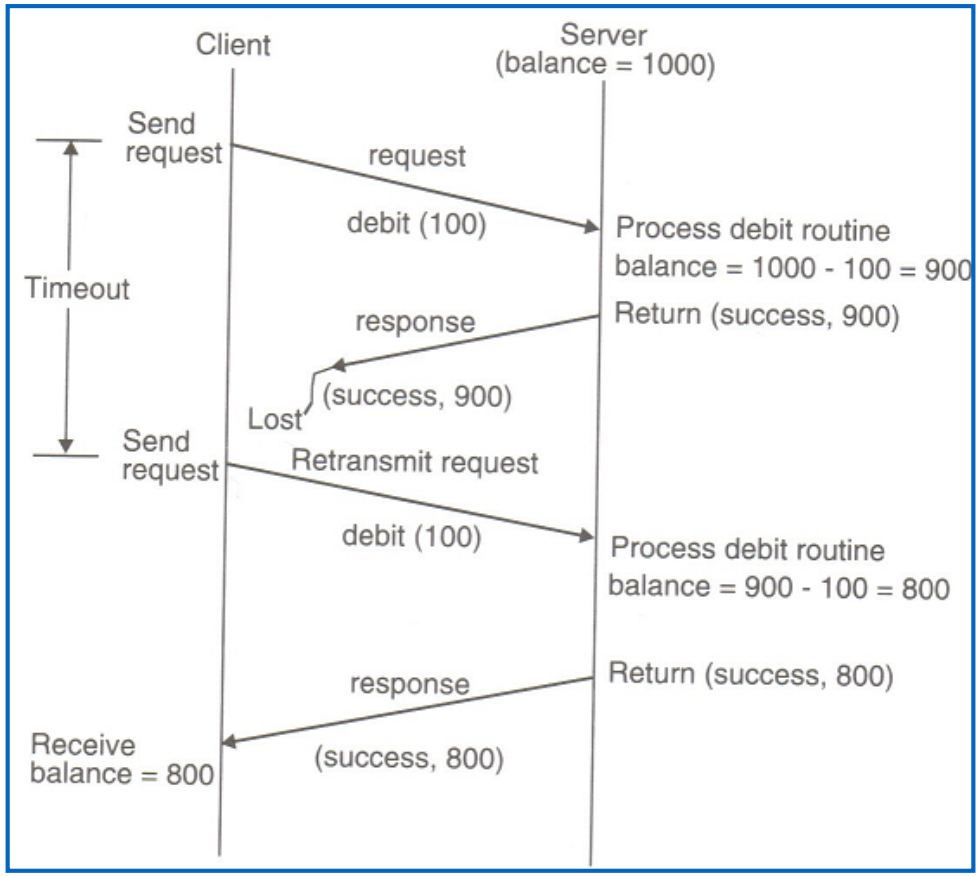
Idempotency

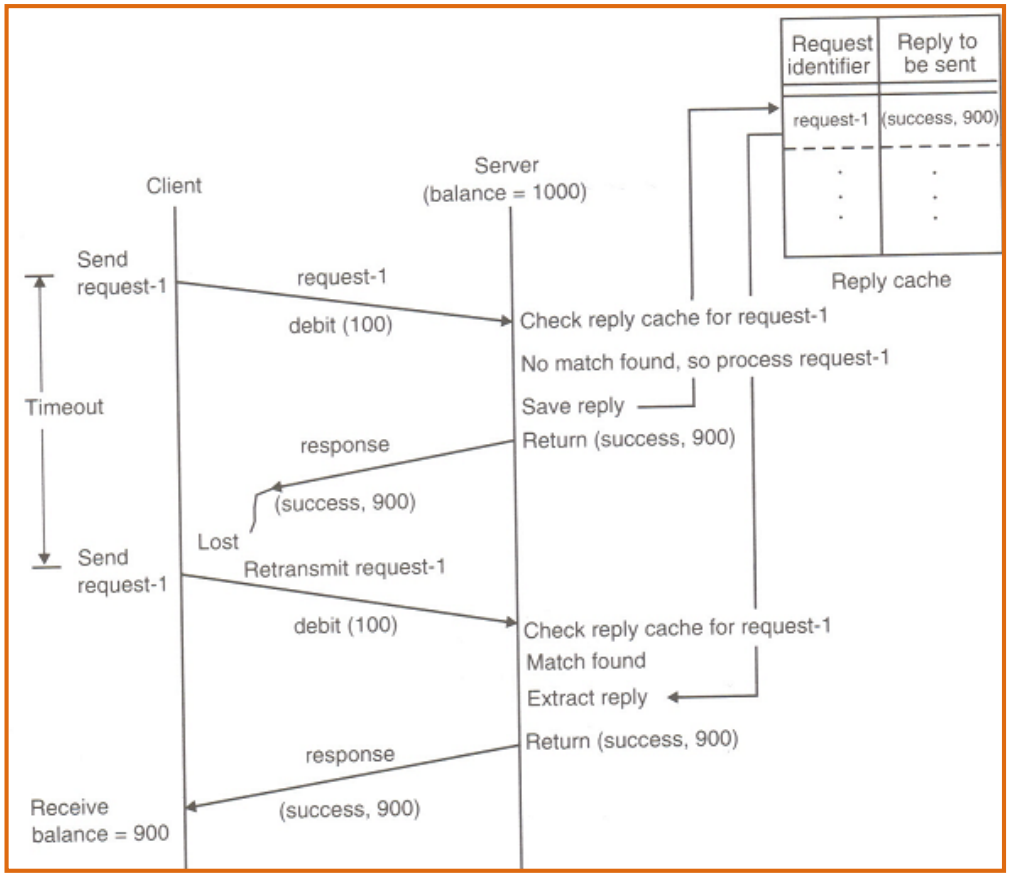
- What is the difference between the following two functions?

```
getSqrt(n)
  {  return sqrt(n)
  }

Debit (amount)
  { if (balance>amount)
  balance= balance-amount
  return ("success", balance)
    else return ("failure", balance)
  }
```

Implementation of Idempotency





– Non-Idempotent

– Adding sequence number with the request message
– Introduction of 'Reply cache'

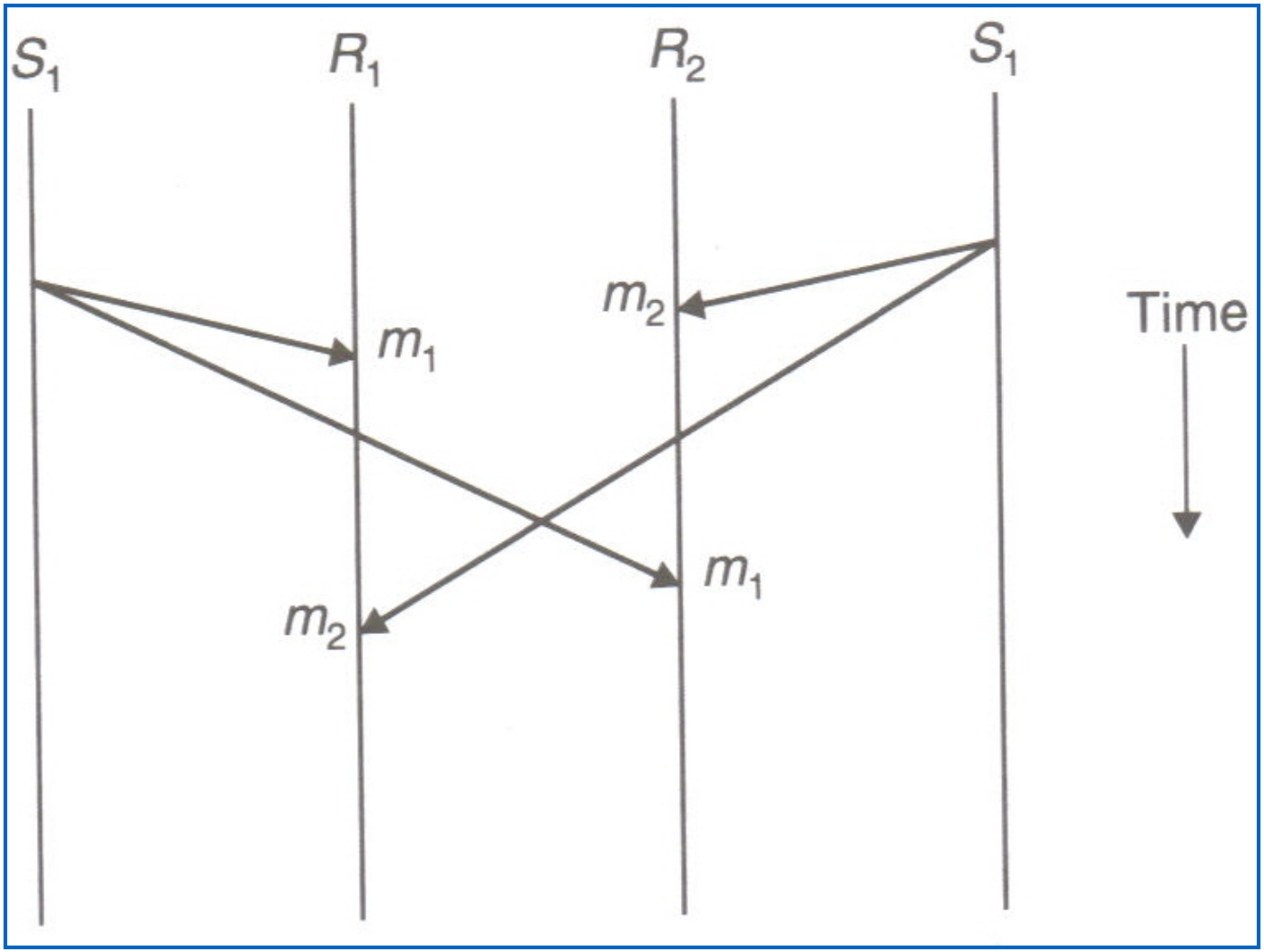
Group Communication

- Group communication:
  - One to many, Many to one, Many to many
- One to many
  - Group management, Group addressing
  - Buffered and unbuffered multicast
  - Send-to-all and Bulletin-Board semantics
  - Flexible reliability in multicast communication
  - Atomic multicast

Many to Many communication

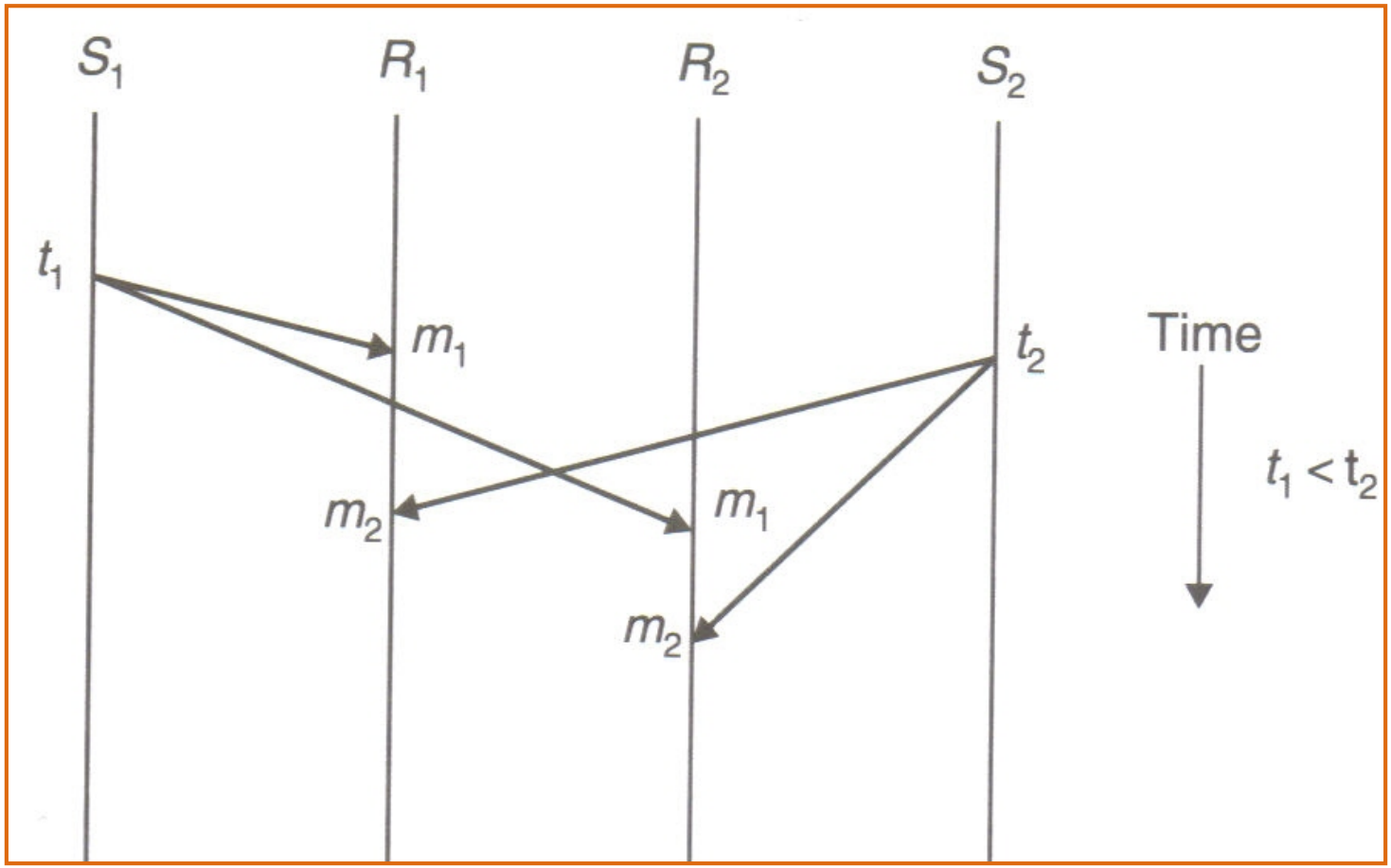
- *Ordered message delivery* is an important issue.
- For example, two server processes are maintaining a single salary database. Two client processes send updates for a salary record. What happen if they reach in different order? (will sequencing of messages help in this case?)

Ordering Semantics



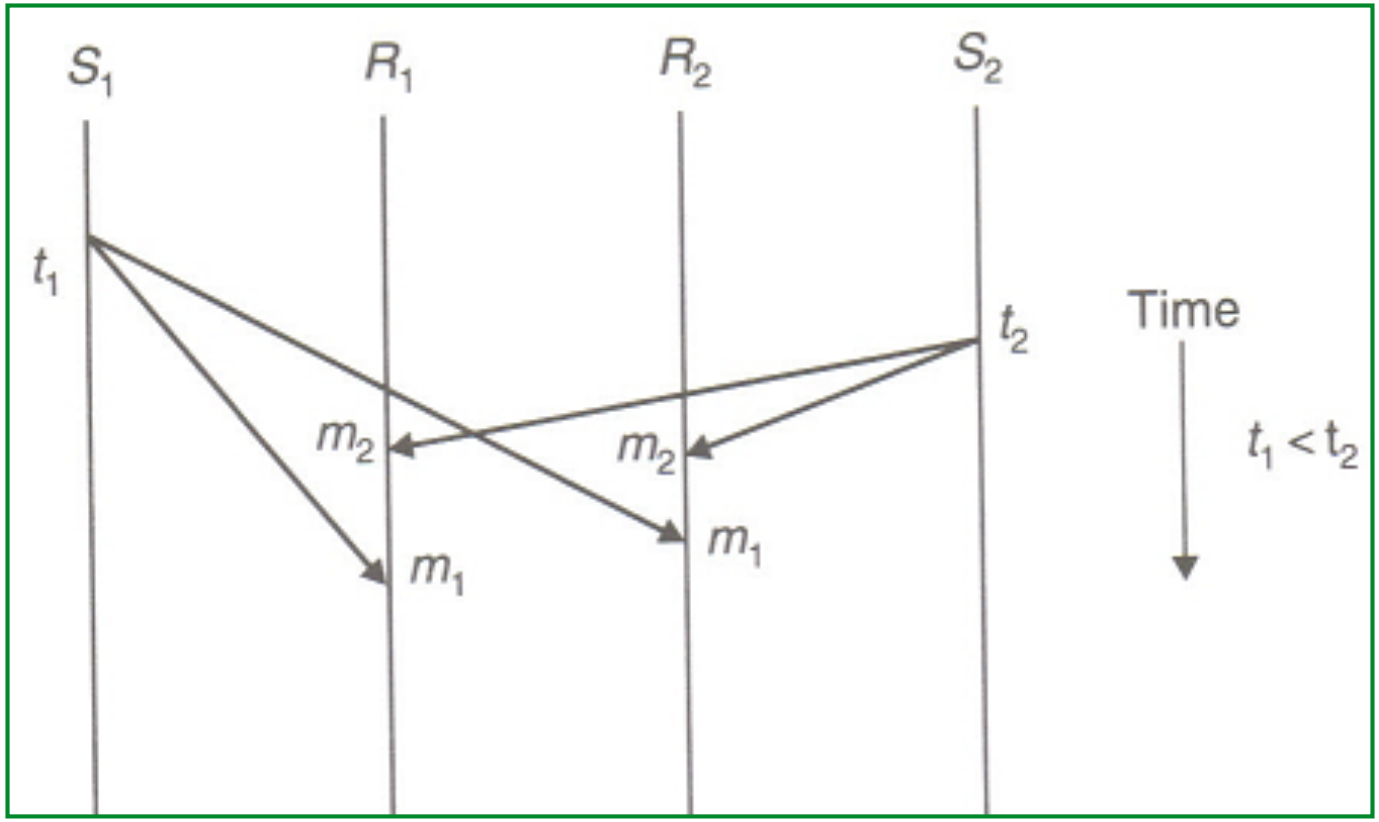
No ordering constraint

Absolute Ordering Semantics



All messages are delivered to all receiver processes in the exact order in which they were sent. Using global timestamp as message identifiers with sliding window protocol

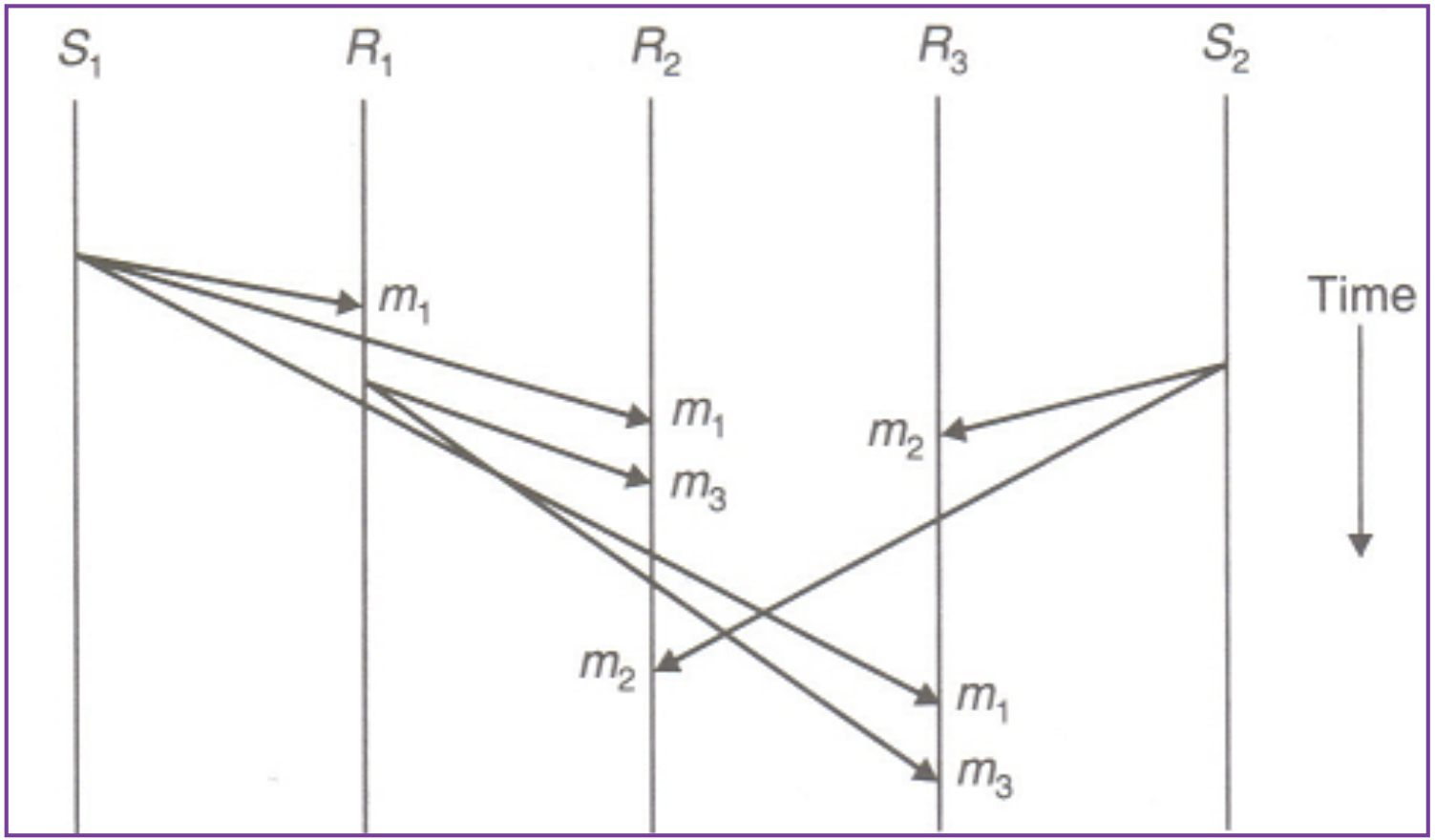
Consistent Ordering Semantics



All messages are delivered to all receivers in the same order. However, this order may be different from the order in which messages were sent.

Causal Ordering Semantics

3rd round m2 is held back.



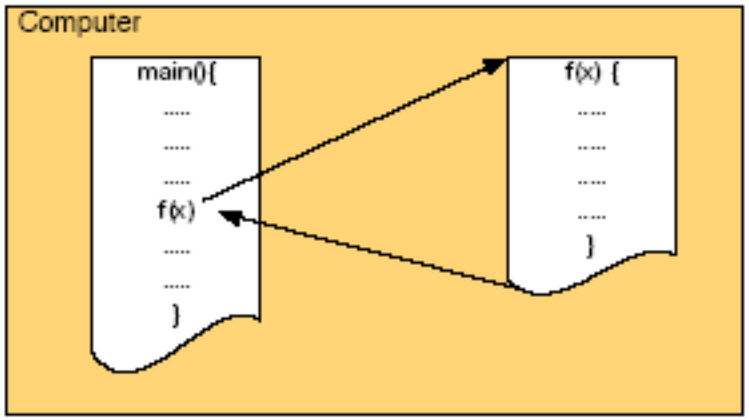
If the event of sending one message is causally related to the event of sending another message, the two messages are delivered to all receivers in correct order. Two message sending events are said to be causally related if they are correlated by the *happened-before* relation.

Remote Procedure Call (RPC)

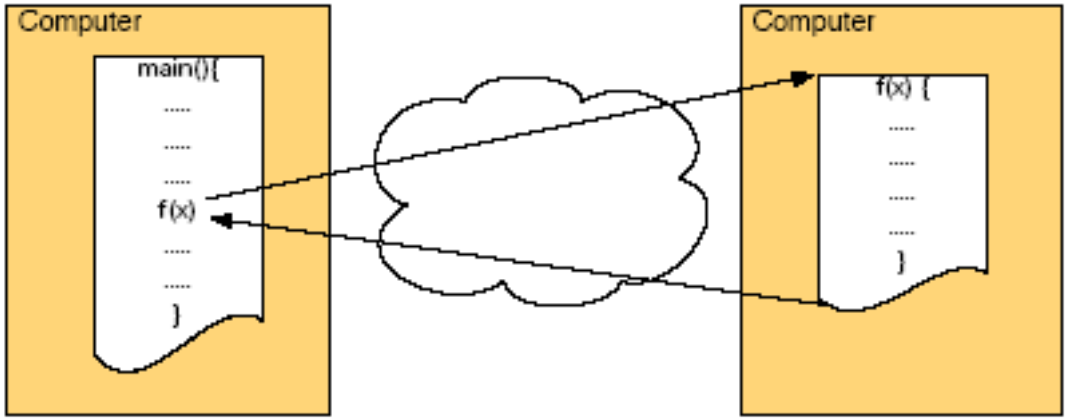
Features of RPC

- Simple call syntax
- Familiar semantics
- Specification of a well defined interface
- Ease of use
- Generality
- Efficiency

Local and Remote Procedure Call

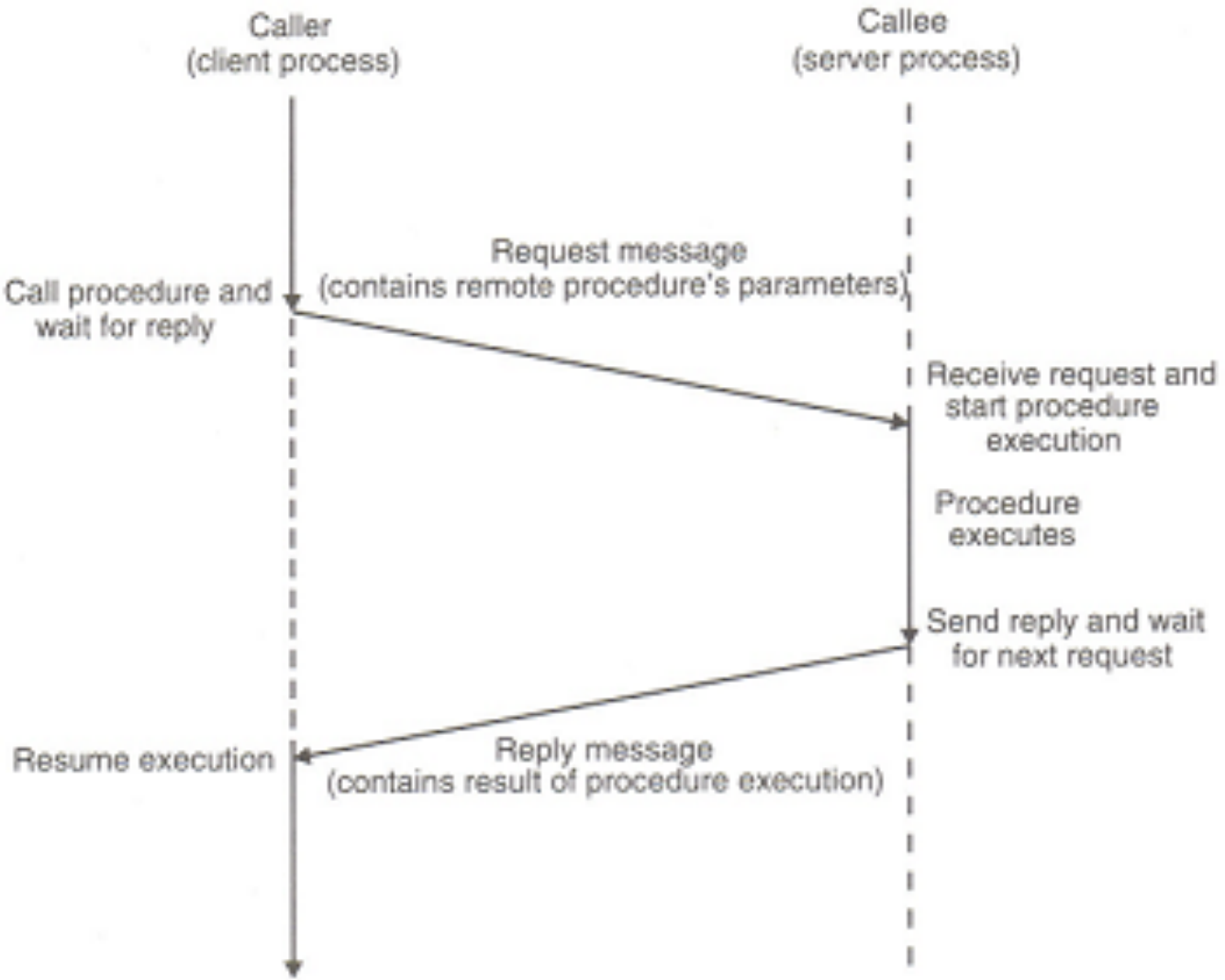


Local Procedure Call



Remote Procedure Call

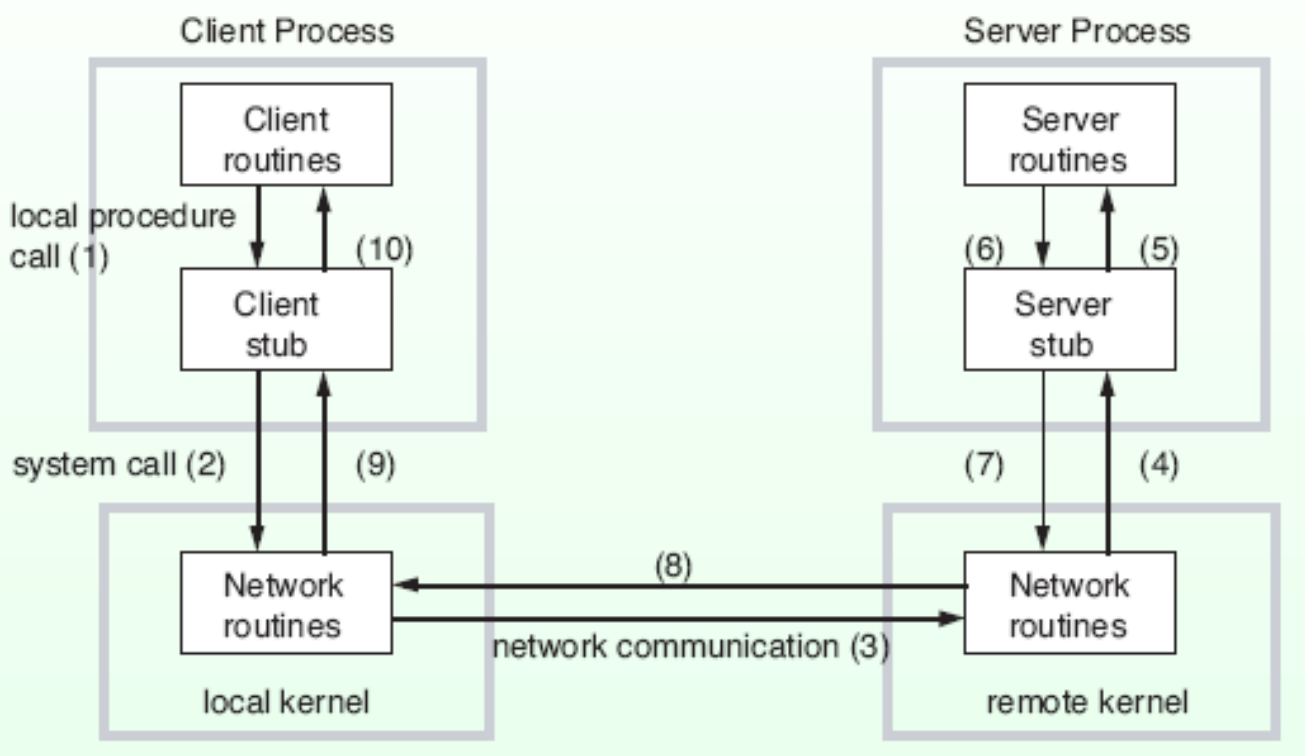
A Typical Model of RPC



Implementing RPC mechanism

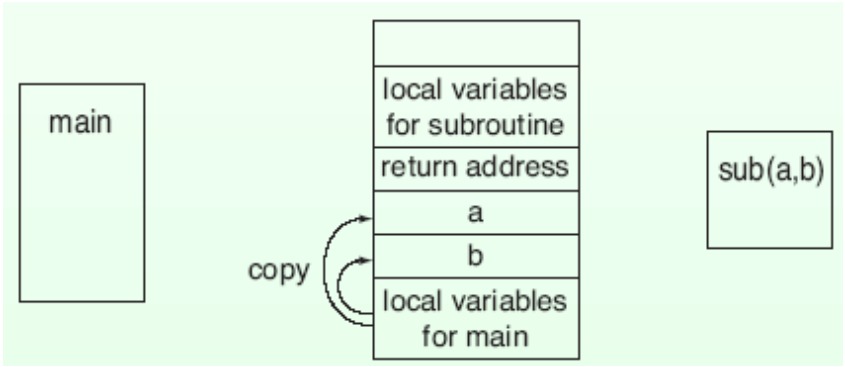
RPC implementation involves five elements:

- The client
- The client stub
- The RPCRuntime
- The server stub
- The server

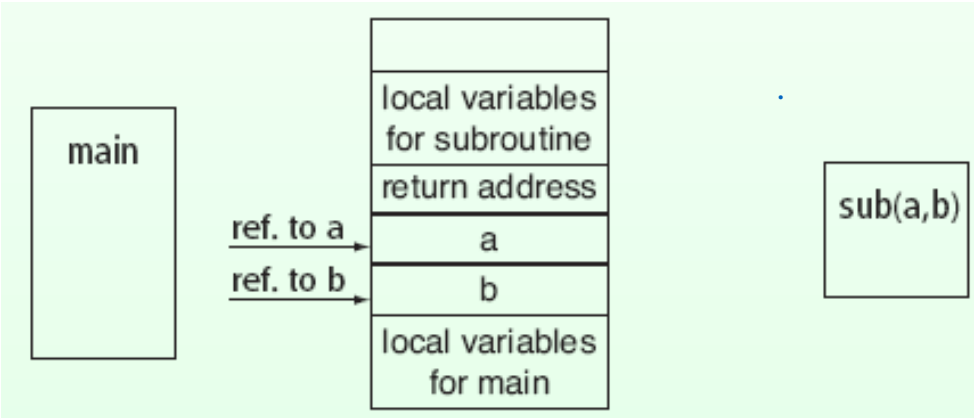


Parameter Passing Mechanisms

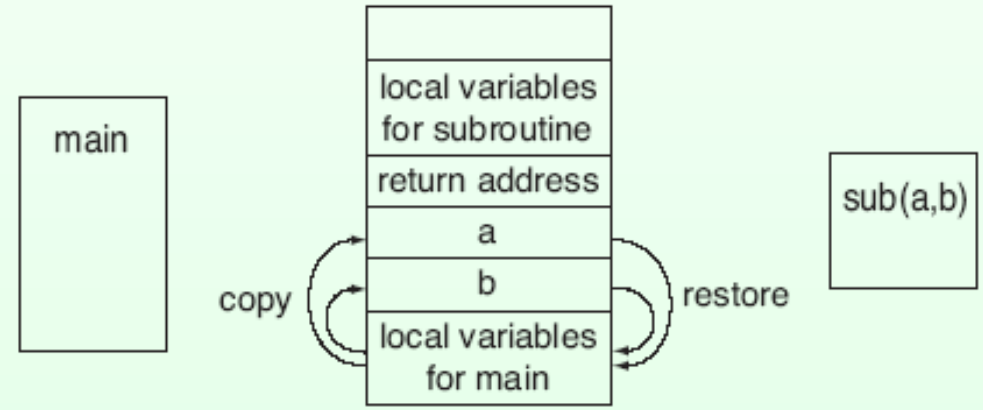
- When a procedure is called, parameters are passed to the procedure as the arguments. There are three methods to pass the parameters.



- call-by-value

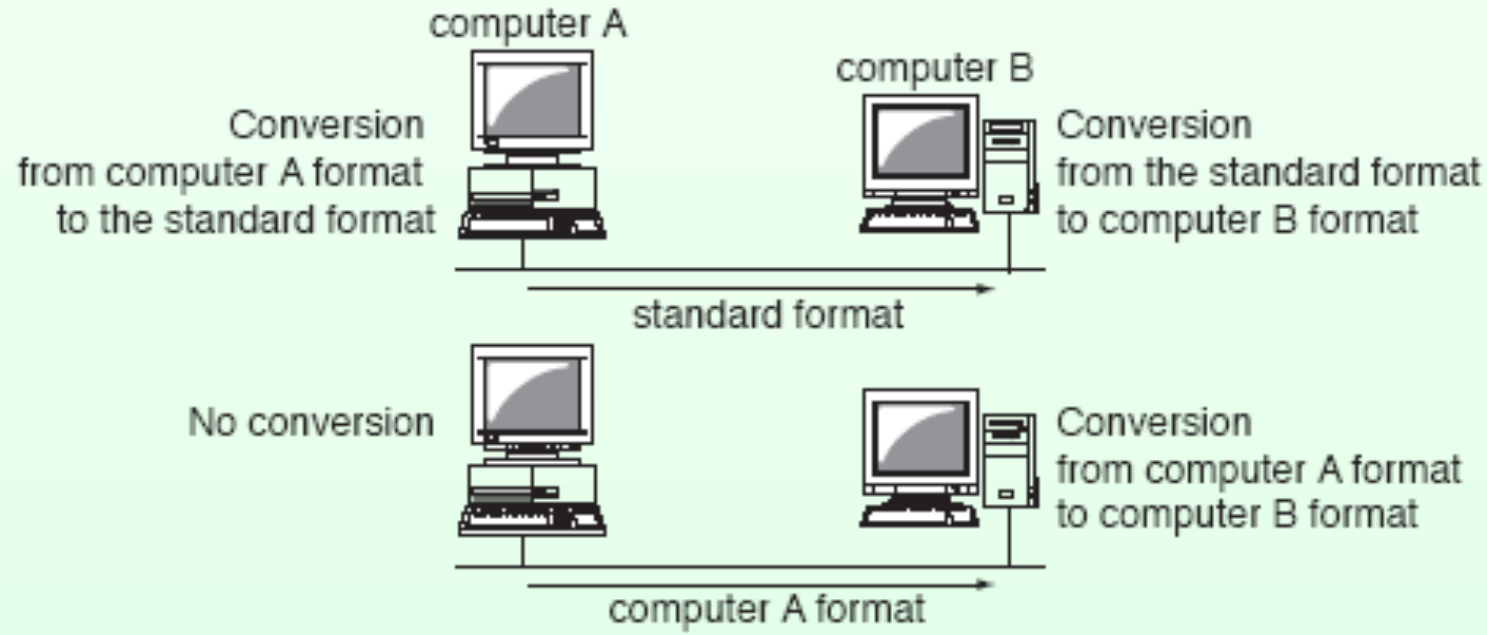


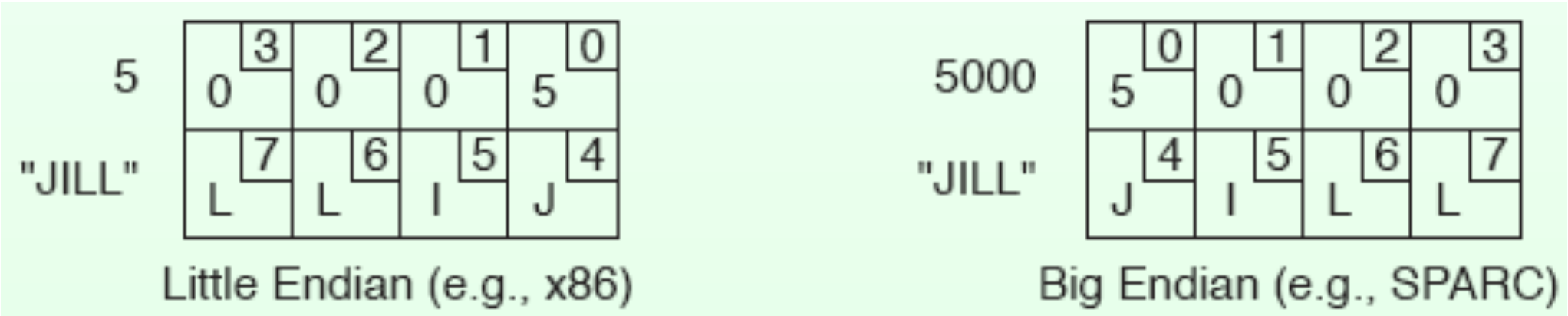
- call-by-reference



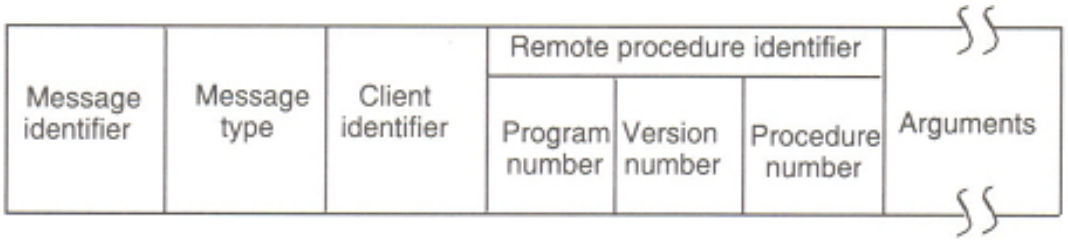
- call-by-copy/restore

Transmit over the network

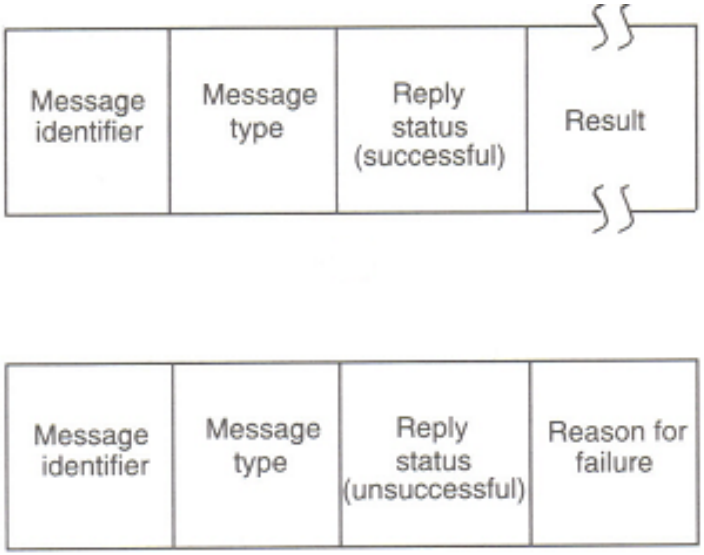




RPC Messages



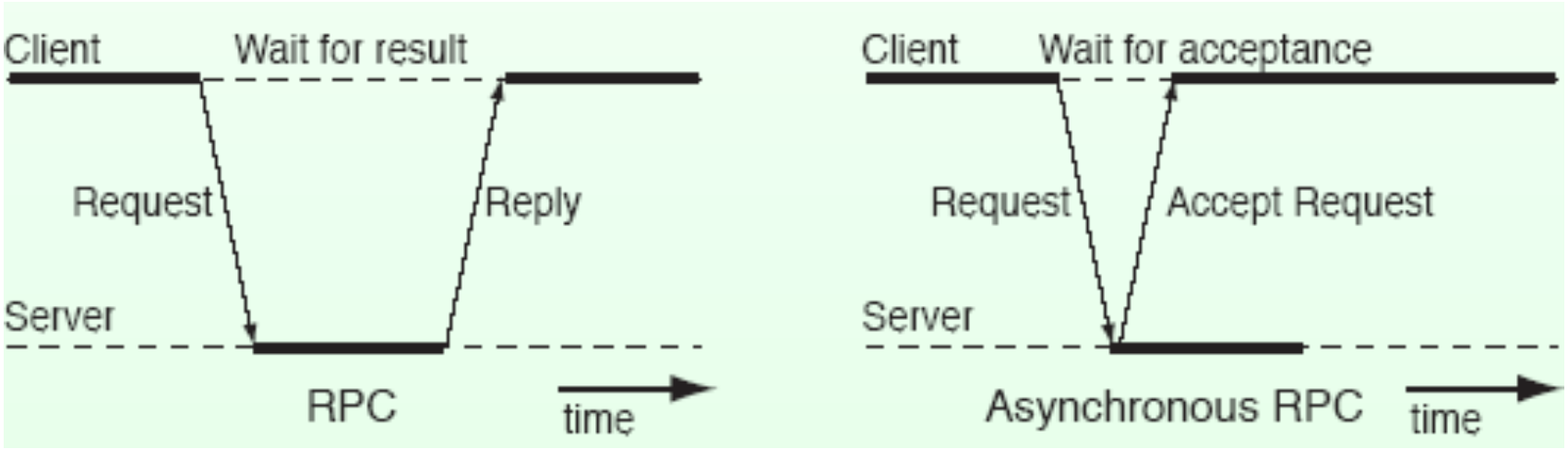
A typical RPC Call message format



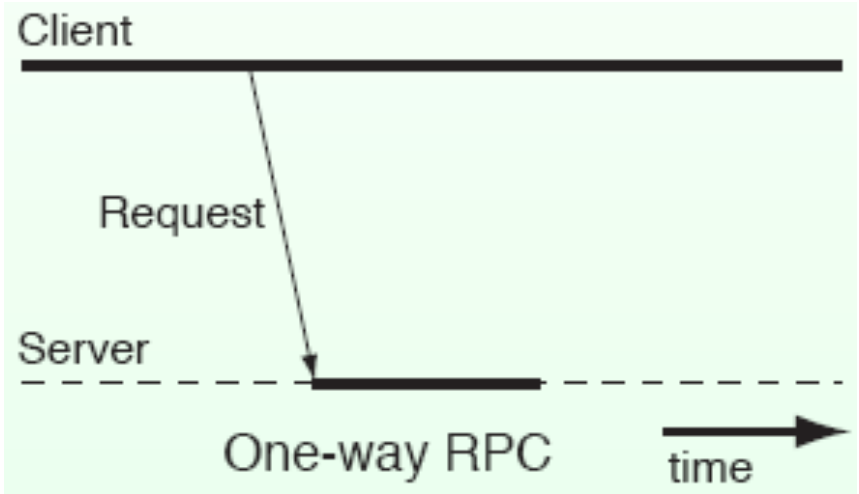
A typical RPC Reply message format

(successful and unsuccessful)

Variations of RPC



Asynchronous RPC



One-way RPC

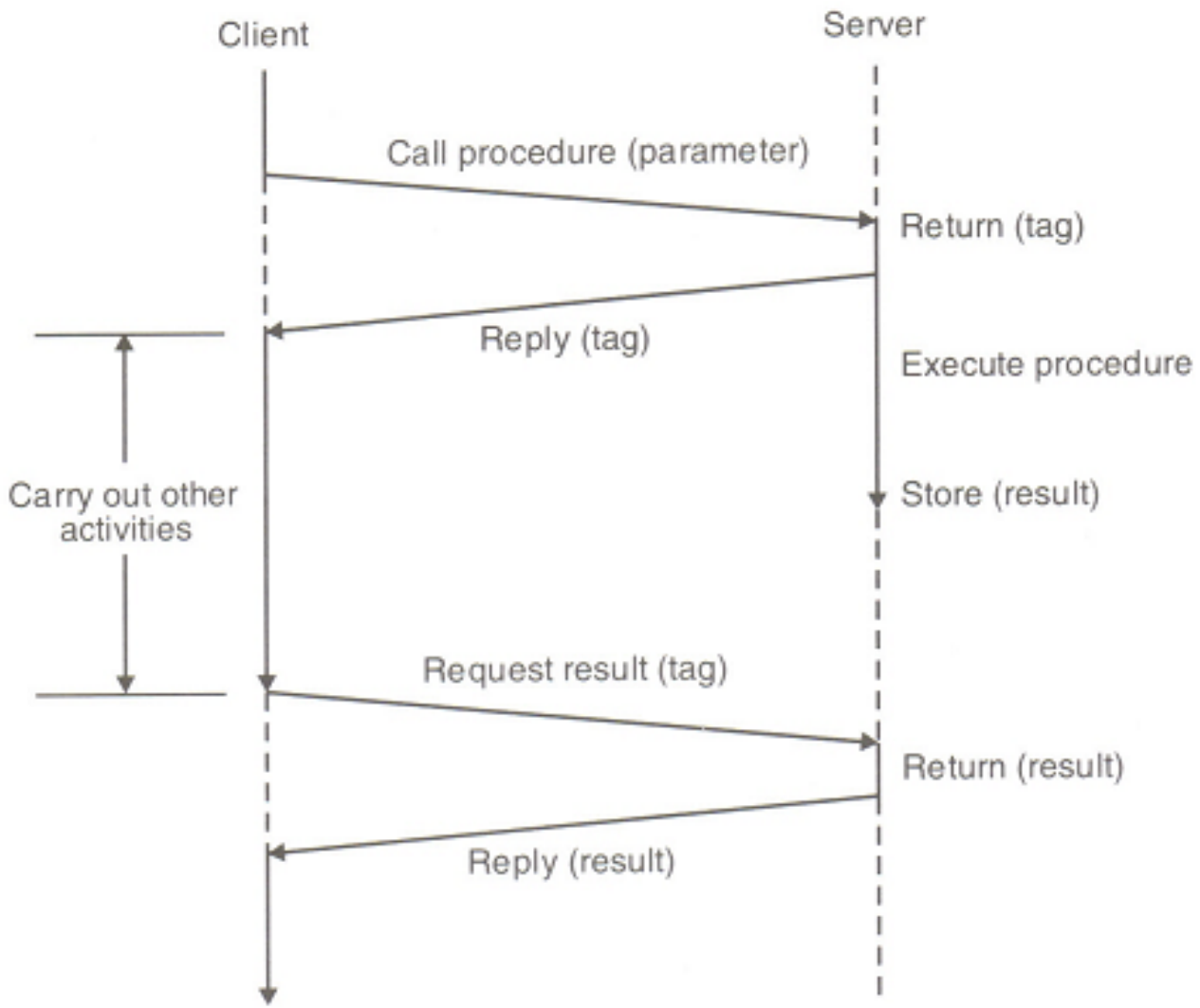
Optimizations for better Performance

- Six different ways:
  - Concurrent access to multiple servers
  - Serving multiple requests simultaneously
  - Reducing per-call workload of servers
  - Reply caching of idempotent remote procedures
  - Proper selection of timeout values
  - Proper design of RPC protocol specification

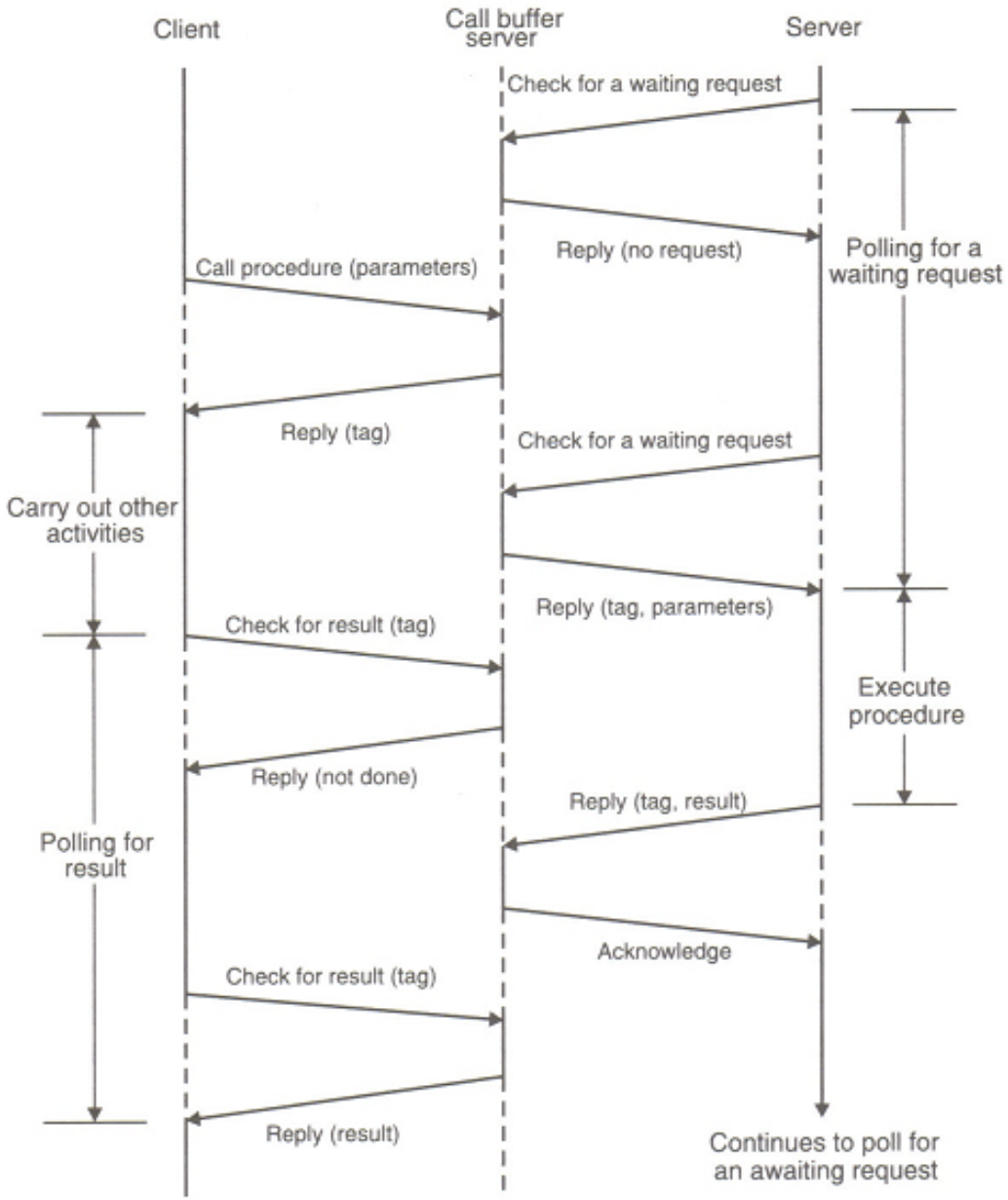
Concurrent Access to Multiple Servers

- One of the following three may be adopted:
  - Threads
  - Early reply approach [Wilbur and Bacarisse]
  - Call buffering approach [Gimson]

Early Reply Approach



Call Buffering Approach



Serving Multiple Requests Simultaneously

- Following types of delays are common:
  - A server, during the course of a call execution, may wait for a shared resource
  - A server calls a remote function that involves computation or transmission delays
- So the server may accept and process other requests while waiting to complete a request.
- Multiple-threaded server may be a solution.

FLUX: Multithreading

Multithreading

JYGAFY

References

- Birman, K. P. and Renesse, R. V. Reliable Distributed Computing with the ISIS Toolkit. IEEE Computer Society Press, 1994.
- Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2(1), 39-59, 1984.
- Wilbur, S. and Bacarisse, B. *Building distributed systems with remote call*, Software Engineering Journal, 2(5), 148-159, 1987.
- R. Gimson. Call buffering service. Technical Report 19, Programming Research Group, Oxford University, Oxford University, Oxford, England, 1985.

Lab Week 3 Overview

- Write a serial program
- Convert to parallel program using POSIX Threads

Tutorial Week 4 Overview

- Inter-Process Communications
- RPC

Next week: Distributed Memory

- Parallel Computing in Distributed Memory
- Message Passing Library