Communication in Distributed Systems
Outline

• Message Oriented Communication
  ➢ Message-Oriented Transient Communication
  ▪ Message-Oriented Persistent Communication
• Stream-Oriented Communications
• RPC/RMI
Message-Oriented Communication

• RPC/RMI are also known as request/response.
• If we assume that there are no responses, we move into message-oriented communication.
• MOC is typically more decoupled than RPC/RMI.
Middleware Protocols

An adapted reference model for networked communication.
Data Link Layer

Discussion between a receiver and a sender in the data link layer.
Persistence and Synchronicity

With transient communication, discard the message if next hop is down. Persistent communication will store.

– How long should it store the message?
– There is also a spectrum between transient and persistent.

Do we wait for the receiver to actually receive the message?

– Messages can be synchronous
Persistence and Synchronicity in Communication (1)

General organization of a communication system in which hosts are connected through a network.
Persistence and Synchronicity in Communication (2)

a) Persistent asynchronous communication
   - A sends message and continues
   - B is not running
   - B starts and receives message
   - Time

b) Persistent synchronous communication
   - A sends message and waits until accepted
   - Message is stored at B's location for later delivery
   - Accepted
   - B is not running
   - B starts and receives message
   - Time
c) Transient asynchronous communication

(d) Receipt-based transient synchronous communication
e) Delivery-based transient synchronous communication at message delivery

f) Response-based transient synchronous communication
Message-Queuing

- MPI and sockets are both transient models.
- Often it is useful to have persistence, to handle servers being down, network interruptions, etc.
Message-Queuing Model (1)

Four combinations for loosely-coupled communications using queues.
Basic interface to a queue in a message-queuing system.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block.</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue.</td>
</tr>
</tbody>
</table>
### Channels

Some attributes associated with message channel agents.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport type</td>
<td>Determines the transport protocol to be used</td>
</tr>
<tr>
<td>FIFO delivery</td>
<td>Indicates that messages are to be delivered in the order they are sent</td>
</tr>
<tr>
<td>Message length</td>
<td>Maximum length of a single message</td>
</tr>
<tr>
<td>Setup retry count</td>
<td>Specifies maximum number of retries to start up the remote MCA</td>
</tr>
<tr>
<td>Delivery retries</td>
<td>Maximum times MCA will try to put received message into queue</td>
</tr>
</tbody>
</table>
Data Stream (1)

Setting up a stream between two processes across a network.
Data Stream (2)

Setting up a stream directly between two devices.
Data Stream (3)

An example of multicasting a stream to several receivers.
Specifying QoS (1)

<table>
<thead>
<tr>
<th>Characteristics of the Input</th>
<th>Service Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>• maximum data unit size (bytes)</td>
<td>• Loss sensitivity (bytes)</td>
</tr>
<tr>
<td>• Token bucket rate (bytes/sec)</td>
<td>• Loss interval (μsec)</td>
</tr>
<tr>
<td>• Token bucket size (bytes)</td>
<td>• Burst loss sensitivity (data units)</td>
</tr>
<tr>
<td>• Maximum transmission rate (bytes/sec)</td>
<td>• Minimum delay noticed (μsec)</td>
</tr>
<tr>
<td></td>
<td>• Maximum delay variation (μsec)</td>
</tr>
<tr>
<td></td>
<td>• Quality of guarantee</td>
</tr>
</tbody>
</table>

A flow specification.
Specifying QoS (2)

The principle of a token bucket algorithm.
Synchronization Mechanisms (1)

The principle of explicit synchronization on the level data units.
The principle of synchronization as supported by high-level interfaces.
Client-Server TCP

a) Normal operation of TCP.
b) Transactional TCP.
Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active
Client and Server Stubs

Principle of RPC between a client and server program.
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client
Passing Value Parameters (1)

Steps involved in doing remote computation through RPC

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
Passing Value Parameters (2)

<table>
<thead>
<tr>
<th></th>
<th>a)</th>
<th>b)</th>
<th>c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>J</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

(a) Original message on the Pentium
(b) The message after receipt on the SPARC
(c) The message after being inverted. The little numbers in boxes indicate the address of each byte
Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message.

```c
foobar( char x; float y; int z[5] )
{
    ....
}
```

(b) foobar's local variables
- x
- y
- 5
- z[0]
- z[1]
- z[2]
- z[3]
- z[4]

(a)
Asynchronous RPC (1)

a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC
Asynchronous RPC (2)

A client and server interacting through two asynchronous RPCs
/* print a list of primes between 1 and 1000 */
main()
{
    int I, how_many, primes[1000];
    how_many = find_primes(1, 1000, primes);
    for (I = 0; I < how_many; I++)
        printf("%d is prime\n", primes[I]);
}
/* Find all primes between min and max, return them in an array */
int find_primes(int min, int max, int *array)
{
    int I, count = 0;
    for (I = min; I <= max; I++)
        if (isprime(I)) array[count++] = I;
    return count;
}
/* Return TRUE If n is prime */
int isprime(int n)
{
    int I;
    for (I = 2; I*I <= n; I++)
        if ((n % I) == 0) return 0;
    return 1;
}
const MAXPRIMES = 1000;
struct prime_request{
    int min;
    int max;
};

struct prime_result{
    int array<MAXPRIMES>;
};

program PRIMEPROG{
    version PRIMEVERS{
        prime_result FIND_PRIMES(prime_request) = 1;
    } = 1;
} = 0x32345678;
prime_result *find_primes_1(prime_request *request)
{
    static prime_result result;
    static int prime_array[MAXPRIMES];
    int i, count = 0;

    for (i = request->min; i <= request->max; i++)
    {
        if (isprime(i)) prime_array[count++] = i;
    }

    result.array.array_len = count;
    result.array.array_val = prime_array;
    return(&result);
}

int isprime(int n)
{
    int i;
    for (i = 2; i*i <= n; i++)
    {
        if ((n % i) == 0) return 0;
    }
    return 1;
}
#include <rpc/rpc.h>
#include "primes.h"

main(int argc, char *argv[]) {
    int i;
    CLIENT *cl;
    prime_result *result;
    prime_request request;

    if (argc != 4) {
        printf("usage: %s host min max\n", argv[0]);
        exit(1);
    }
    cl = clnt_create(argv[1], PRIMEPROG, PRIMEVERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(2);
    }
    request.min = atoi(argv[2]);
    request.max = atoi(argv[3]);
    result = find_primes_1(&request, cl);
    if (result == NULL) {
        clnt_perror(cl, argv[1]);
        exit(3);
    }
    for (i = 0; i < result->array.array_len; i++)
        printf("%d is prime\n", result->array.array_val[i]);
    printf("count of primes found = %d\n", result->array.array_len);
    xdr_free(xdr_prime_result, result);
    clnt_destroy(cl);
}
Identifying an RPC Service

There is a service registry called the port mapper, which keeps track of all RPC services registered on a machine, and their transport endpoint addresses.

In later implementations the port mapper is called “rpcbind”.

Remote procedure is identified by THREE numbers:

\{program number, version number, procedure number\}

While \{program number, version number\} identify a specific server program, the procedure number identifies a procedure within that program.

The followings are the steps involved in locating an RPC service:

- When server starts, it create a endpoint to accept connection, so it binds an arbitrary port number for this purpose;
- It then send a message to the portmapper registering the service;
- Portmapper add this mapping to its list;
- When client wants to find the service, it ask the portmapper by passing it the program number, version number and protocol it uses;
- Portmapper returns the port number to the client;
- Client call the server procedure by sending an RPC call message to the port it just got from the portmapper;
- The call message contains the serialized arguments and the procedure number.
Identifying an RPC Service

Step 1: ‘This is PRIMEPROG Version 1, I’m using port 1061’

Step 2: ‘Where is PRIMEPROG Version 1?’

Step 3: ‘It’s on port 1061’

Step 4: ‘Call procedure no. 1. Here’s the data ...’

Portmapper

Server

Port 1061

Port 111

Client

Prog  Vers  Port

___  ___  ___
Five different classes of failures in RPC systems:

– **Client Cannot Locate the Server**
  - Server might be down or new server with old client
  - return -1 for error is not good enough because the return value might be -1 (e.g. adding 7 to -8)
  - raise an exception (so write your own SIGNOSERVER)

– **Lost Request Messages**
  - easiest to deal with, just have the kernel start a timer when sending the request, and resend it if time-out.

– **Lost Reply Messages**
  - use timer again, but this time you cannot tell whether the reply is lost or the server is just slow
  - idempotent transactions
  - reading the first 1024 byte of a file is idempotent;
  - transferring 1 million $ from a bank account is non-idempotent
  - solution 1: construct all requests in an idempotent way
  - solution 2: have the client’s kernel assign each request a seq. number
- Server Crashes

- In the 2nd case, the system has to report failure back to the client
- In the 3rd case, it can just retransmit the request
  - at least-once semantics
  - at most-once semantics
  - exactly once semantics
Client Crashes

- unwanted computation is called orphan
- solution 1: extermination -- keeping log for every request
- solution 2: reincarnation -- divide time into sequentially numbered epochs. Client broadcasts a message to all machines declaring a new epoch. All remote computations are killed.
- solution 3: gentle reincarnation -- same as 2, but remote computations are killed only if they cannot find their owner
- solution 4: expiration -- each RPC is given a standard time T to do a job, if it cannot finish within T, it have to ask the for another quantum. After the client crash, it wait for T unit of time to make sure that all orphans are gone, then it reboots.

note: none of these methods are desirable. Worst yet, killing an orphan may have unforeseen consequences. For example, suppose that an orphan has obtained locks on one or more files or database records. If the orphan is killed suddenly, these locks may remain forever. Also, an orphan may have already made entries in various remote queues to start up other processes at some future time, so even killing the orphan may not remove all traces of it.
Distributed Objects

Common organization of a remote object with client-side proxy.

- Client machine
  - Client
  - Proxy
  - Client OS
  - Client invokes a method
  - Same interface as object

- Server machine
  - Server
  - Skeleton
  - Server OS
  - Skeleton invokes same method at object

Network

Marshalled invocation is passed across network

like client stub

like server stub