Distributed Systems (COMP9243)

Lecture 4: Communication

Slide 1

① Communication in a Distributed System
   ➔ Shared memory vs message passing
   ➔ Communication modes
② Communication Abstractions

Slide 2

Why Communication?

Cooperating processes need to communicate.
   ➔ For synchronisation and control
   ➔ To share data

Slide 3

In a Non-Distributed System:
Two approaches to communication:
   ➔ Shared memory

Slide 4

Shared Memory:

[Diagram showing shared memory between two processes]
In a Non-Distributed System:
Two approaches to communication:
→ Shared memory
  • Direct memory access (Threads)
  • Mapped memory (Processes)
→ Message passing

Communication in a Distributed System
Previous slides assumed a uniprocessor or a multiprocessor.
In a distributed system (multicomputer) things change:
Shared Memory:
→ There is no way to physically share memory
Message Passing:
→ Over the network
→ Introduces latencies
→ Introduces higher chances of failure
→ Heterogeneity introduces possible incompatibilities
MESSAGE PASSING

Basics:
- send()
- receive()

Variations:
- Connection oriented vs Connectionless
- Point-to-point vs Group
- Synchronous vs Asynchronous
- Buffered vs Unbuffered
- Reliable vs Unreliable
- Message ordering guarantees

Data Representation:
- Marshalling
- Endianness

COUPLING

Dependency between sender and receiver

Temporal: do sender and receiver have to be active at the same time?

Spatial: do sender and receiver have to know about each other? explicitly address each other?

Semantic: do sender and receiver have to share knowledge of content syntax and semantics?

Platform: do sender and receiver have to use the same platform?

Tight vs Loose coupling: yes vs no

COMMUNICATION MODES

Data-Oriented vs Control-Oriented Communication:

Data-oriented communication
- Facilitates data exchange between threads
- Shared address space, shared memory & message passing

Control-oriented communication
- Associates a transfer of control with communication
- Active messages, remote procedure call (RPC) & remote method invocation (RMI)

Synchronous vs Asynchronous Communication:

Synchronous
- Sender blocks until message received
- Often sender blocked until message is processed and a reply received
- Sender and receiver must be active at the same time
- Receiver waits for requests, processes them (ASAP), and returns reply
- Client-Server generally uses synchronous communication

Asynchronous
- Sender continues execution after sending message (does not block waiting for reply)
- Message may be queued if receiver not active
- Message may be processed later at receiver’s convenience

When is Synchronous suitable? Asynchronous?
Transmit vs Persistent Communication:

**Transient**
- Message discarded if cannot be delivered to receiver immediately
- Example: HTTP request

**Persistent**
- Message stored (somewhere) until receiver can accept it
- Example: email

Coupling?

Direct-Addressing vs Indirect-Addressing Communication:

**Direct-Addressing**
- Message sent directly to receiver
- Example: HTTP request

**Indirect-Addressing**
- Message not sent to a particular receiver
- Example: broadcast, publish/subscribe

Coupling?

Provider-Initiated vs Consumer-Initiated Communication:

**Provider-Initiated**
- Message sent when data is available
- Example: notifications

**Consumer-Initiated**
- Request sent for data
- Example: HTTP request

Combinations:

- **Persistent Asynchronous**
  - Message can be sent only if B is running

- **Transient Asynchronous**
  - Request Received

- **Persistent Synchronous**
  - Start processing request

- **Transient Synchronous (Receipt Based)**
  - Request Received

- **Transient Synchronous (Delivery Based)**
  - Accepted

- **Transient Synchronous (Response Based)**
  - Accepted

Examples?
**COMMUNICATION ABSTRACTIONS**

Abstractions above simple message passing make communication easier for the programmer.

Provided by higher level APIs

- Message-Oriented Communication
- Request-Reply, Remote Procedure Call (RPC) & Remote Method Invocation (RMI)
- Group Communication
- Event-based Communication
- Shared Space

**MESSAGE-ORIENTED COMMUNICATION**

Communication models based on message passing

Traditional `send()/receive()` provides:

- Asynchronous and Synchronous communication
- Transient communication

What more does it provide than `send()/receive()`?

- Persistent communication (Message queues)
- Hides implementation details
- Marshalling

**EXAMPLE: MESSAGE PASSING INTERFACE (MPI)**

- Designed for parallel applications
- Makes use of available underlying network
- Tailored to transient communication
- No persistent communication
- Primitives for all forms of transient communication
- Group communication

MPI is BIG. Standard reference has over 100 functions and is over 350 pages long!

**EXAMPLE: MESSAGE QUEUING SYSTEMS**

- Designed for parallel applications
- Makes use of available underlying network
- Tailored to transient communication
- No persistent communication
- Primitives for all forms of transient communication
- Group communication

MPI is BIG. Standard reference has over 100 functions and is over 350 pages long!
Slide 21

Provides:
- Persistent communication
- Message Queues: store/forward
- Transfer of messages between queues

Model:
- Application-specific queues
- Messages addressed to specific queues
- Only guarantee delivery to queue. Not when.
- Message transfer can be in the order of minutes

Examples:
- IBM MQSeries, Java Message Service, Amazon SQS, Advanced Message Queuing Protocol, MQTT, STOMP

Very similar to email but more general purpose (i.e., enables communication between applications and not just people)

Slide 22

**REQUEST-REPLY COMMUNICATION**

Request:
- a service
- data

Reply:
- result of executing service
- data

Requirement:
- Message formatting
- Protocol

Slide 23

**EXAMPLE: REMOTE PROCEDURE CALL (RPC)**

Idea: Replace I/O oriented message passing model by execution of a procedure call on a remote node (BN84):
- Synchronous - based on blocking messages
- Message-passing details hidden from application
- Procedure call parameters used to transmit data
- Client calls local "stub" which does messaging and marshalling

Confusing local and remote operations can be dangerous, why?

Slide 24

Remember Erlang client/server example?:

```erlang
% Client code using the increment server
client (Server) ->
  Server ! {self (), 10},
  receive
  {From, Reply} -> io:format (“Result: “w’n”, [Reply])
  end.

% Server loop for increment server
loop () ->
  receive
  {From, Msg} -> From ! {self (), Msg + 1},
  loop ()
  stop -> true
  end.

% Initiate the server
start_server() -> spawn (fun () -> loop () end).
```
This is what it’s like in RPC:

% Client code
client (Server) ->
    register(server, Server),
    Result = inc (10),
    io:format ("Result: ~w~n", [Result]).

% Server code
inc (Value) -> Value + 1.

Where is the communication?

RPC Implementation:

1. client calls client stub (normal procedure call)
2. client stub packs parameters into message data structure
3. client stub performs send syscall and blocks
4. kernel transfers message to remote kernel
5. remote kernel delivers to server stub, blocked in receive()
6. server stub unpacks message, calls server (normal proc call)
7. server returns to stub, which packs result into message
8. server stub performs send() syscall
9. kernel delivers to client stub, which unpacks and returns

Example client stub in Erlang:

% Client code using RPC stub
client (Server) ->
    register(server, Server),
    Result = inc (10),
    io:format ("Result: ~w~n", [Result]).

% RPC stub for the increment server
inc (Value) ->
    server ! {self (), inc, Value},
    receive
      {From, inc, Reply} -> Reply
    end.
Example server stub in Erlang:

```erlang
% increment implementation
c:inc (Value) -> Value + 1.

% RPC Server dispatch loop
server () ->
  receive
    {From, inc, Value} ->
      From ! {self(), inc, inc(Value)}
  end,
  server().
```

Parameter marshalling:
- stub must pack ("marshal") parameters into message structure
- message data must be pointer free
  (by-reference data must be passed by-value)
- may have to perform other conversions:
  - byte order (big endian vs little endian)
  - floating point format
  - dealing with pointers
  - convert everything to standard ("network") format, or
  - message indicates format, receiver converts if necessary
- stubs may be generated automatically from interface specs

Examples of RPC frameworks:
- SUN RPC (aka ONC RPC): Internet RFC1050 (V1), RFC1831 (V2)
  - Based on XDR data representation (RFC1014)(RFC1832)
  - Basis of standard distributed services, such as NFS and NIS
- Distributed Computing Environment (DCE) RPC
- XML (data representation) and HTTP (transport)
  - Text-based data stream is easier to debug
  - HTTP simplifies integration with web servers and works through firewalls
  - For example, XML-RPC (lightweight) and SOAP (more powerful, but often unnecessarily complex)
- Many More: Facebook Thrift, Google Protocol Buffers RPC, Microsoft .NET

Sun RPC Example:
Run example code from website
Sun RPC - interface definition:

```c
program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1; /* proc num = 1 */
        string STR_DATE(long) = 2; /* proc num = 2 */
    } = 1; /* version = 1 */
} = 0x31234567; /* prog num */
```

Sun RPC - client code:

```c
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
...}
main(int argc, char **argv) {
    CLIENT *cl; /* RPC handle */
    ...
    cl = clnt_create(argv[1], DATE_PROG, DATE_VERS, "udp");
    lresult = bin_date_1(NULL, ... = str_date_1(lresult, cl);
    printf("time on host %s = %s", server, *sresult);
    clnt_destroy(cl); /* done with the handle */
}
```

Sun RPC - server code:

```c
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */

long * bin_date_1() {
    static long timeval; /* must be static */
    long time(); /* Unix function */
    timeval = time((long *) 0);
    return(&timeval);
}
char ** str_date_1(long *bintime) {
    static char *ptr; /* must be static */
    char *ctime(); /* Unix function */
    ptr = ctime(bintime); /* convert to local time */
    return(&ptr); /* return the address of pointer */
}
```

**ONE-WAY (ASYNCHRONOUS) RPC**

- When no reply is required
- When reply isn’t needed immediately (2 asynchronous RPCs - deferred synchronous RPC)
REMOTE METHOD INVOCATION (RMI)

Like RPC, but transition from the server metaphor to the object metaphor.

Why is this important?

- RPC: explicit handling of host identification to determine the destination
- RMI: addressed to a particular object
- Objects are first-class citizens
- Can pass object references as parameters
- More natural resource management and error handling
- But still, only a small evolutionary step

TRANSPARENCY CAN BE DANGEROUS

Why is the transparency provided by RPC and RMI dangerous?

- Remote operations can fail in different ways
- Remote operations can have arbitrary latency
- Remote operations have a different memory access model
- Remote operations can involve concurrency in subtle ways

What happens if this is ignored?

- Unreliable services and applications
- Limited scalability
- Bad performance

See “A note on distributed computing” (Waldo et al. 94)

GROUP-BASED COMMUNICATION

Two kinds of group communication:

- Broadcast (message sent to everyone)
- Multicast (message sent to specific group)

Used for:

- Replication of services
- Replication of data
- Service discovery
- Event notification

Issues:

- Reliability
- Ordering

Example:

- IP multicast
- Flooding
**EXAMPLE: Gossip-Based Communication**

Technique that relies on epidemic behaviour, e.g. spreading diseases among people.

Variant: rumour spreading, or gossiping.

- When node $P$ receives data item $x$, it tries to push it to arbitrary node $Q$.
- If $x$ is new to $Q$, then $P$ keeps on spreading $x$ to other nodes.
- If node $Q$ already has $x$, $P$ stops spreading $x$ with certain probability.

Analogy from real life: Spreading rumours among people.

---

**Event-Based Communication**

- Communication through propagation of events
- Generally associated with publish/subscribe systems
- Sender process publishes events
- Receiver process subscribes to events and receives only the ones it is interested in.
- Loose coupling: space, time
- Example: OMG Data Distribution Service (DDS), JMS, Tibco

---

**Shared Space Communication**

Example: Distributed Shared Memory:

```
CPU 1  CPU 2  CPU 3  CPU 4
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

Shared global address space

---

**Tuple Space**

Example: Tuple Space:

```
A Write A
B Write B
T Read T
```

Insert a copy of A
Insert a copy of B
Look for tuple that matches T
Return C (and optionally remove it)

Coupling?
**Reading List**

**Slide 45**

**Implementing Remote Procedure Calls** A classic paper about the design and implementation of one of the first RPC systems.

**Homework**

**RPC:**
- Do Exercise Client server exercise (Erlang) Part B

**Slide 46**

**Synchronous vs Asynchronous:**
- Explain how you can implement synchronous communication using only asynchronous communication primitives.
- How about the opposite?