CS 5523 Operating Systems: Application-Level Communications

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu, Dr. Palden Lama for providing their slides.
What are the Problems?

- Process A wants to send process B a message
  - “Tongping Liu, Assistant Professor, CS Dept UTSA, x5550”

- Application-level agreement
  - Which part is what (name, title, contact)

- Network: actual message transmission
  - Issues with addressing, performance, scalability, reliability, security
Outline

- Fundamentals
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous

- Models for application communications
  - **RPC**: remote procedure call
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication
C/S Comm.: Request Protocol (R)

- If service
  - does not have output parameters and
  - does not have a return type

client may not want to wait for server to finish.
C/S Comm.: Request-Reply protocol (RR)

- If client expects results from a server
  - Client requests service execution from server through a request message, and
  - Delivery of service results in a reply message

- Most client-server interactions are built on RR protocol

Diagram:
- Client sends request message, blocked until server responds.
- Server receives request, executes operation, and sends reply.
- Client receives reply and continues.
Request-Reply-Acknowledge Protocol (RRA)

- In addition to RR protocol, client sends acknowledgement after it received reply
- Acknowledgement sent asynchronously

Client

<table>
<thead>
<tr>
<th>send(...)</th>
<th>receive(...)</th>
<th>send(...)</th>
</tr>
</thead>
</table>

Server

| receive(...) | exec op; |

| send(...) | receive(...) |

history
What may go wrong?

- Request lost
- Server down
- Reply lost
Typical Failure Handling in RR

- **Client side:** request retry with time out
  - Client keeps local copy of request
  - If *timeout* and no reply received, it *resends* request
  - After N attempts, *assumes* that server has failed

- **Server side:** receives a request multiple times
  - *Re-execute & resend* results; or
  - *Store requests & results:* filter duplicated requests and re-transmit results

*How long should servers keep the requests&results?*
Should Servers Re-Do Operations?

- **Idempotent** operations: which can be performed repeatedly with the same effect.
  - suppose $x$ is input message $\rightarrow f(f(x)) = f(x)$
  - No state needs to maintain on the server

- Are the following operations idempotent?
  - HTTP GET ...
  - UNIX file operations: read, write etc.

  - yes
  - NO
Server Invocation Semantics in RR

<table>
<thead>
<tr>
<th>Retransmit request message</th>
<th>Duplicate filtering</th>
<th>Re-execute procedure or retransmit reply</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute procedure</td>
<td>At-least-once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most-once</td>
</tr>
</tbody>
</table>
Server Invocation Semantics (cont.)

- **Maybe**: if no reply, the client does not know if method was executed or not
- **At least once**: will guarantee that invocation be carried out at least once, but possibly more
- **At most once**: Will guarantee that messages has been sent out at most once, but possibly none at all
  - Detect duplicated requests with sequence numbers
  - **No guarantees**: When a server crashes, the client gets no help and no promises about what happened

- Local invocation: **exactly once** - ideal case
Types of Communications

- **Asynchronous communication**
  - Sender continues immediately after it has submitted the request (unblocked, need a local buffer at the sender)

- **Synchronous communication**
  - Sender blocks until the sender receives an OK to continue; *where the OK may come?*
Types of communications (cont.)

**Persistent vs. Transient**

- **Persistent**: A message is stored at a communication server as long as it takes to deliver it (e.g., e-mail).
- **Transient**: A message is stored as long as sender and receiver are working at the same time (TCP, UDP, IP routing)
### Type of Communications (cont.)

<table>
<thead>
<tr>
<th>Persistent</th>
<th>Asynchronous</th>
<th>Synchronous at Submission, delivery, after service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message-oriented middleware (MOM)</td>
<td><strong>Asynchronous</strong></td>
<td><strong>Synchronous at Submission, delivery, after service</strong></td>
</tr>
<tr>
<td>• Processes send each other messages (queued)</td>
<td>• Sender does not need to wait for immediate reply</td>
<td>• Middleware often ensures fault tolerance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transient</th>
<th>Client/Server, RPC, TCP</th>
<th>Client/Server, RPC, TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Client and server have to be active at time of communication</td>
<td>• Client issues request and blocks until it receives reply</td>
<td>• Server essentially waits only for incoming requests, and subsequently processes them</td>
</tr>
</tbody>
</table>

**Drawbacks of synchronous communication**

- Client cannot do any other work while waiting for reply
- Failures have to be handled immediately: the client is waiting
- The model may simply not be appropriate (mail, news)
a) Persistent asynchronous communication (e.g., email)

b) Persistent synchronous communication
c) Transient asynchronous communication (e.g., UDP)
d) Receipt-based transient synchronous communication (e.g., TCP)
Persistence and Synchronization 3

e) Delivery-based **transient synchronous** communication at message delivery (e.g., asynchronous RPC)

f) Response-based **transient synchronous** communication (RPC)
Outline

- **Fundamentals**
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous

- **Models for application communications**
  - RPC: remote procedure call
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication
High-Level Communication Models

- RPC: remote procedure call
- Message-oriented communication
- Stream-Oriented communication
- Multicast communication
Remote Procedure Calls

- Network communication:
  - Send and receive are not transparent
  - Application developers are good at "procedure call" model

- In 1984, Birrell and Nelson allows programs to call procedures located on remote machines (RPC)
  - RPC is integrated into programming languages
  - Like centralized computing, by allowing remote services to be called as procedures

- Issues: How to pass parameters, semantics in face of errors, failures, etc…

RPC is a high-level model for client-server communication
Consider a call in C: \texttt{result = add(para\_x, para\_y)}

Caller pushes parameters onto the stack: \textbf{in one process}

Stack before (a) and while (b) the called procedure is active is shown below
Parameters Passing in Local Calls

- **Call-by-value**: the parameter value copied to the stack. Modifications do not affect the calling side.

- **Call-by-reference**: the address of the parameter is pushed onto the stack (for example, pointers in C; obj ref in Java). Any modification affects the variable at the calling side.

- **Call-by-copy/restore**: Copy the variable first to the stack (as in call-by-value), and then copy back after the call, overwriting the caller’s original value.
  - In many cases, the same behavior as “call-by-reference”
Remote Procedure Call (RPC)

Client would like to use its own functions as well as the ones in the server. How?

- myAdd();
- mySub();
- myMax();
- myMin();

- magicAdd();
- magicSub();
- magicMax();
- magicMin();
Stubs at Client and Server Sides

- Make remote procedure call look like local call
- So that the client can call myAdd() and magicAdd() in the same way

```
result1 = myAdd(para_x, para_y);
result2 = magicAdd(para_x, para_y);
```

**Client stub** for `magicAdd()` packs the parameters into a message and call OS (through `send` primitive). It will then block itself (through `receive` primitive).

**Server stub** for `magicAdd()` unpacks the parameters from the message and calls the local procedure in a usual way. When completes, the server stub will pack the result and return it to the client.
RPC Steps

A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub.
2. The client stub builds a message and calls the local operating system.
3. The client’s OS sends the message to the remote OS.

   4. The remote OS gives the message to the server stub.
   5. The server stub unpacks the parameters and calls the server.
   6. The server does the work and returns the result to the stub.
   7. The server stub packs it in a message and calls its local OS.
   8. The server’s OS sends the message to the client’s OS.

9. The client’s OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Sounds simple but there are several issues…
Pack value parameters into a message and send it to the server. Would it be that easy?
Big Endian and Little Endian

- **Big Endian:**
  - Store the most significant byte in the smallest address.

- **Little Endian:**
  - Store the *least* significant byte in the smallest address.

0x 90AB12CD
Problem 1: different data representations

A process on an Intel machine send a message of an integer and four-character string ("5, JILL") to another process on a Sun SPARC machine.

(a) original message on Intel (x86, Little Endian)

(b) receipt message on SPARC (Big Endian) "5000, JILL"

(c) simple reverse: message after converted "5, LLIJ"

The number is reversed by the string is not...
Approaches for Exchanging Information

**How people from different countries talk to each other?**

- With listener’s language
  - English, Chinese, Indian
  - ... //hundreds

- Common language
  - English

**How can clients make servers on different machines understand them?**

- Clients send information in servers’ data representation

- A common external data representation
  - Language defined representation
  - e.g., CORBA CDR

- External **self-descriptive** data representation
  - e.g., XML, Web Services
Problem 2: Un/Marshaling

- More than just wrapping parameters into a message
- Client and server machines may have different data representations (think of byte ordering)
- Client and server have to agree on a standard representation (e.g., external data representation (XDR))
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, and transform them into machine-dependent representations.
Other Problems in Parameter Passing

- **Passing reference parameters**
  - Pointers, and in general, reference parameters are passed with considerable difficulty

- **Solutions**
  - Forbid reference parameters
  - Copy the entire data structure (e.g. an entire array may be sent if the size is known). In the case of the server input parameter, it does not need to be copied back.
  - How to handle complex data structures (e.g. general graphs)?
Parameter Passing Semantics in RPC

**RPC parameter passing:**
- RPC assumes *copy in/copy out* semantics: while procedure is executed, nothing can be assumed about parameter values.
- RPC assumes *all data that is to be operated on is passed by parameters*. Excludes passing *references to data*.

**Conclusion:** full access transparency cannot be realized.

**Observation:** If we introduce a *remote reference*, access transparency can be enhanced:
- Remote reference offers unified access to remote data.
- Remote references can be passed as parameter in RPCs.
RPC in Practice

- **Essence:** developers concentrate on client- & server-specific codes, while RPC (generators / libraries) do the rest (actual communication)
- Client **stub** and server **stub** communicate over the network to perform RPC
Generate Stubs

- Hiding a remote procedure call requires that the caller and callee agree on **interface**
  - the format of the messages
  - the representation of simple data structures (no complicated data)
  - the transport layer protocol (Connection-oriented or not)

- Using **Interface Definition Language (IDL)** to specify an interface: services provided by servers
  - Compile IDL to generate client & server stubs
DEC RPC Interface

- Type/data definitions (like C).
- Component is described as a PROGRAM
  - Procedures have a result type, a parameter list and a number,
  - Procedure can be called remotely

- Used by client or server directly:
  - Locating servers: static vs. dynamic binding
  - Choosing a transport protocol.
  - Authentication and security.
  - Invoking RPCs dynamically.

- Used by stubs for:
  - Generating unique message IDs.
  - Sending messages.
  - Maintaining message history
RPC Example

/* person.x */
const NL=64;
enum sex_type {
    FEMALE = 1,MALE = 2};
struct Person {
    string first_name<NL>;
    string last_name<NL>;
    sex_type sex;
    string city<NL>;
};

program PERSONPROG {
    version PERSONVERS {
        void PRINT(Person)=0;
        int STORE(Person)=1;
        Person LOAD(int)=2;
    } = 1;
    } = 1234567;

- Type/data definitions (like C).
- Component is described as a PROGRAM
  - Procedures have a result type, a parameter list and a number,
  - Procedure can be called remotely
Sun RPC Example

- person.x
- person_clnt.c
- person_svr.c
- client.c
- person.h
- server.c
- person_xdr.c
- rpcgen
- librpc.a
- C Compiler, Linker
- Client
- Server

- includes
- generates
- reads
Sun RPC Server Implementation

/* server.c */

void * print_0(Person *argp,
              struct svc_req * rqstp)
{
    static char * result;
    printf("%s %s\n%sn\n\n",
           argp->first_name,
           argp->last_name,
           argp->city);
    return((void *) &result);
}
Sun RPC Client

/* client.c */
print_person(char * host, Person * pers) {
    CLIENT *clnt;
    clnt = clnt_create(host, PERSONPROG, PERSONVERS, "udp");
    if (clnt == (CLIENT *) NULL) {
        exit(1);
    }
    if (print_0(pers, clnt)==NULL) {
        exit(1);
    }
    if (print_0(pers, clnt)==NULL)
        clnt_perror(clnt, "call failed");
    clnt_destroy(clnt);
}

How to locate an RPC server that can execute a given procedure in a network?
Binding a Client to a Server

- Registration of a server makes it possible for a client to locate the server and bind to it
- Locate the server’s machine ➔ locate server

Diagram:
- Client machine
- Directory machine
- Server machine
- 1. Register end point
- 2. Register service
- 3. Look up server
- 4. Ask for end point
- 5. Do RPC
Binding

- **Server**
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder

- **Client**
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

- **Performance issues**
  - Exporting and importing incurs overhead
  - Binder can be a bottleneck (Use multiple binders)
  - Binder can do load balancing
Steps in 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"
Lightweight RPCs

- Many RPCs occur between client and server on same machine
  - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server exports interface to remote procedures
- Client on same machine imports interface
- OS kernel creates data structures including an argument stack shared between Server and Client
Lightweight RPCs (cont.)

- **RPC execution**
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Client thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
Other RPC Models

- **Asynchronous RPC**
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later

- **Deferred-synchronous RPC**
  - Use two asynchronous RPCs
  - Client needs a reply but can’t wait for it; server sends reply via another asynchronous RPC

- **One-way RPC**
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).
Traditional RPC vs. Asynchronous RPC

- Try to get rid of the strict request-reply behavior, and let the client continue without waiting for an answer from the server.

Traditional synchronous RPC

asynchronous RPC
server stub immediately
sends a reply to client.
Deferred Synchronous RPC

- Client can also do a (non)blocking poll at the server to see whether results are available.
- Through two asynchronous RPCs
One-Way RPC

- Client does not know if the request is accepted or not (ch 8 fault tolerance)
RPC Limitations

- Parameters passed by values only and pointer values are possibly not allowed.

- Speed: remote procedure calling (and return) time (i.e., overheads) can be significantly (1 - 3 orders of magnitude) slower than that for local procedure.

  ➢ This may affect real-time design and the programmer should be aware of its impact.
RPC Limitations

- Failure: RPC is more vulnerable to failure (since it involves communication system, another machine and another process).

  - The programmer should be aware of the call semantics, i.e. programs that make use of RPC must have the capability of handling errors that cannot occur in local procedure calls.
Design Issues

- Exception handling
  - Necessary because of possibility of network and nodes failures;
  - RPC uses return value to indicate errors;

- Transparency
  - Syntactic → achievable, exactly the same syntax as a local procedure call;
  - Semantic → impossible because of RPC limitation: failure (similar but not exactly the same);
Outline

Fundamentals

- Client/Server communication protocols
  - Request vs. Request-reply vs. Request-reply-acknowledge
- Invocation semantics
  - Exact once vs. at least once vs. at most once
- Communication types
  - Transient vs. persistent
  - Synchronous vs. asynchronous

Models for application communications

- RPC: remote procedure call (inherent synchronous)
- Message-oriented communication
- Stream-Oriented communication
- Multicast communication
Message-Oriented Communication

- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere MQ
Transient Messaging (transport-level sockets)

- Berkeley sockets

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Server

Client

Synchronization point

Communication
Message-Oriented Middleware

- **Asynchronous persistent** communication through support of middleware-level queues.
  - Queues correspond to buffers at communication servers
  - Communicated processes may NOT be active at the same time

<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>
Message-Oriented Middleware (cont.)

- Four different loosely-coupled communications

- Diagrams showing different states of senders and receivers:
  - (a) Sender running, Receiver running
  - (b) Sender running, Receiver passive
  - (c) Sender passive, Receiver running
  - (d) Sender passive, Receiver passive
General Architecture of Message Queue
Message-Queuing System with Routers
Message Broker

- Message queuing systems assume a common messaging protocol: **all applications agree on message format** (i.e., structure and data representation)

- **Message broker**: Centralized component that takes care of application heterogeneity in an MQ system:
  - **Transforms** incoming messages to target format
  - Very often acts as an **application gateway**
  - May provide **subject-based** routing capabilities
  ⇒ **Enterprise Application Integration**
Message Broker (cont.)

Pull/push models
Stream-Oriented Communication

- Applications
  - Video-conference
  - Online multimedia player/viewer (YouTube, CNN Live etc.)

- Support for continuous media
- Streams in distributed systems
- Stream management
Continuous Media

- **Observation:** All discussed communication facilities are based on a **discrete** or **time-independent** exchanging of information.

- **Continuous media:** Characterized by the fact that values are **time dependent**:
  - Audio
  - Video
  - Animations
  - Sensor data (temperature, pressure, etc.)
Transmission of Continuous Media

- Different timing guarantees: 3 types of transmission

  - **Asynchronous**: no restrictions with respect to *when* data is to be delivered

  - **Synchronous**: define a maximum end-to-end delay for individual data packets

  - **Isochronous**: define a maximum and minimum end-to-end delay (*jitter* is bounded)
Definition: A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.

Common stream characteristics:
- Streams are unidirectional
- A single source, and one or more sinks
- Often, either the sink and/or source is a wrapper around hardware (e.g., CD device, TV monitor, dedicated storage)

Two types of streams:
- Simple: single flow of data, e.g., audio or video
- Complex: multiple data flows, e.g., stereo audio or combination audio/video
Streams and QoS

Streams are all about **timely** delivery of data

**Quality of Service (QoS) specifications**

- The required **bit rate** at which data should be transported.
- The **maximum delay** until a session has been set up (i.e., when an application can start sending data).
- The **maximum end-to-end delay** (i.e., how long it will take until a data unit makes it to a recipient).
- The maximum delay variance, or **jitter**.
- The **maximum round-trip delay**.
Enforce QoS
Enforce QoS: Example

- Network-level tools: prioritize packets (differentiated services)
- Use **buffers** to reduce jitter
Enforce QoS: Example

**Interleaving**: multiple samples in a single packet: the effects of packet loss → spread the samples

(a)

(b)
Application-level Multicast

- Multicasting: multiple receivers
  - Organize nodes of a distributed system into an overlay network and use that network to disseminate data
  - Network routers are not a part of groups

- Epidemic-based data dissemination
Epidemic Algorithms

**Basic idea:** rapidly propagate information among a network using only local information (Infect all nodes AFAP)
- Update operations are initially performed at a few nodes
- A node passes its updated state to a limited number of neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every node

**Propagation models**
- **Anti-entropy (entropy: lack of order or predictability):** chooses another node at random, and exchanges state differences, leading to identical states at both
- **Gossiping:** A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
Anti-Entropy Propagation Model

- Node P selects node Q from the system at random.
- Schemes to propagates updates:
  - **Push**: P only sends its updates to Q.
  - **Pull**: P only retrieves updates from Q.
  - **Push-Pull**: P and Q exchange mutual updates (after which they hold the same information).
- Susceptible: not-infected; Removed: updated but not willing to spread.
- Observation: pure push-based is not effective:
  - Updates can be propagated by infected nodes.
  - When many nodes are infected, the chance to pickup a susceptible node is very small.
Anti-Entropy Propagation Model

- **Pull-based**: works better when many nodes are infected. Spreading is triggered by susceptible nodes.

- **Push-pull**: best approach of anti-entropy.
  - $O(\log(N))$ rounds to disseminate updates to all $N$ nodes
    (\textit{round} = when every node as taken the initiative to start an exchange).
  - Propagating is fast and scalable
Gossiping or Rumor Spreading

A server $S$ having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, $S$ stops contacting other servers with probability $1/k$.

Gossiping in reality:

- Bob has hot news, he may tell Alice
- Alice will spread the gossip to her friends as well.
- When Alice tell Charlie about this news, if Charlie said that he already heard this news, thus Alice will be disappointed and may stop to spread this news 😞
Gossiping or Rumor Spreading

- Gossiping is an excellent way of rapidly spreading news.
- Gossiping alone is not enough for full propagation.

If $k = 4$, then $\ln(s)$ is -4.97, then $s$ is less than 0.007. Less than 0.7% of the nodes remain susceptible.
Gossiping vs. Epidemic

- **Advantage of Gossiping:**
  - Scalability.

- **Epidemic algorithm:**
  - Advantage: good for spreading updates
  - But spreading the removal is hard since the deletion of a data item destroys all information on that item: when a node receive old copies of the data item, they will think of it as updates on something that it didn’t have before.
Deleting Values

- **Fundamental problem:** cannot remove an old value from a server and expect the removal to propagate.

- **Death certificate:** Removal has to be registered as a special update

- **When to remove a death certificate?**
  - Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
  - Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)
Summary

Fundamentals

- Client/Server communication protocols
  - Request vs. Request-reply vs. Request-reply-acknowledge
- Invocation semantics
  - Exact once vs. at least once vs. at most once
- Communication types
  - Transient vs. persistent
  - Synchronous vs. asynchronous

Communication models

- RPC: remote procedure call
- Message-oriented communication
- Stream-Oriented communication
- Multicast communication