Chapter 5
Remote Invocation

Gandeva Bayu Satrya, ST., MT.
Telematics Labz.
Informatics Department
Telkom University
@2014

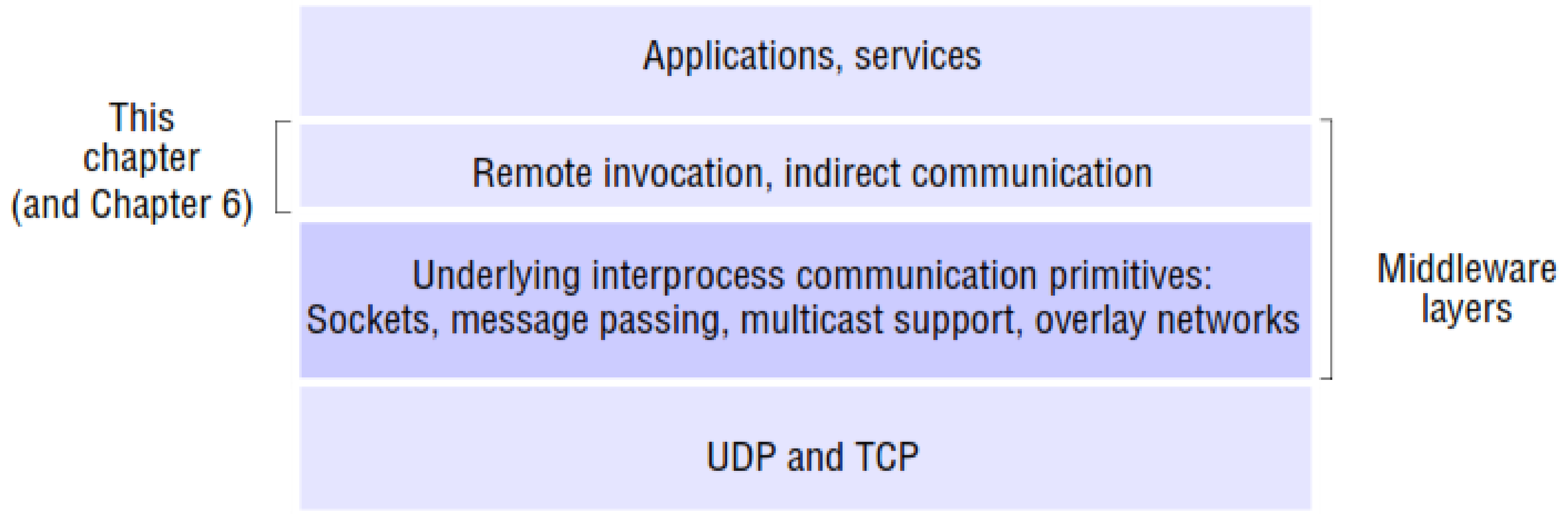
Outline Today

Chapter 5 – Remote Invocation.

Chapter 6 – Indirect Communication.

- Request-Reply Protocols
- RPC
- Remote Method Invocation
- Group Communication
- Shared Memory Approach

Introduction



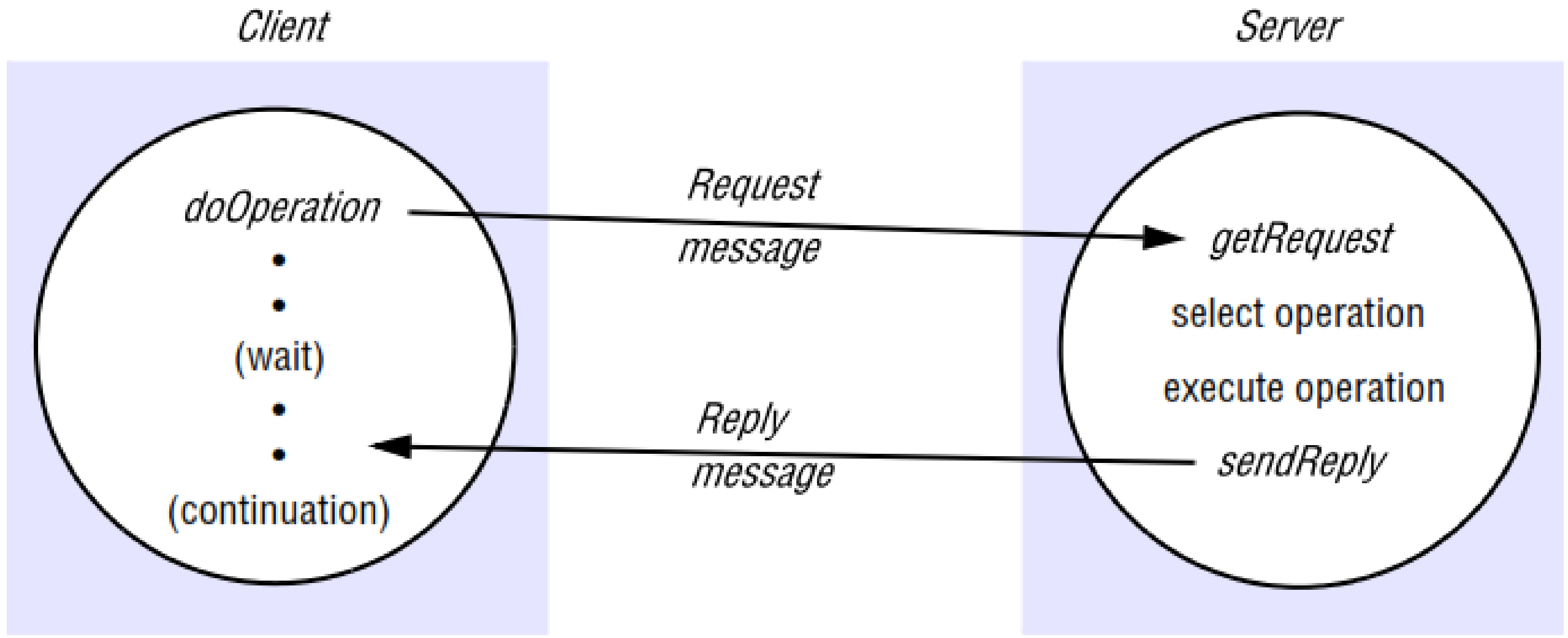
1. Request-Reply Protocols

- **Request-Reply Communication** is synchronous because the client process blocks until the reply arrives from the server.

- It can also be Reliable because the reply from the server is effectively an acknowledgement to the client.

- The **client-server** exchanges are described in the following paragraphs in terms of the send and receive operations in the Java API for UDP datagrams, although many current implementations use TCP streams.

Request-Reply Communication



1. Request-Reply Protocols (con't)

a) The Request-Reply Protocol : doOperation, getRequest and sendReply
b) Message identifiers : requestId and identifier
c) Failure model : omission failures
d) Timeouts : return immediately
e) Discarding duplicate request messages : receive it more than once
f) Lost reply messages : has already sent
g) History : contains a record
h) Styles of exchange protocols : R-RR-RRA
i) Use of TCP streams to implement the request-reply protocol : reliably
j) HTTP : An example of a request-reply protocol
k) Message contents : message body

2. RPC

- The concept of a **Remote Procedure Call (RPC)** represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming.
- That is, achieving a high level of distribution transparency.
- in **RPC**, procedures on remote machines can be called as if they are procedures in the local address space.

A client and server through two asynchronous RPCs

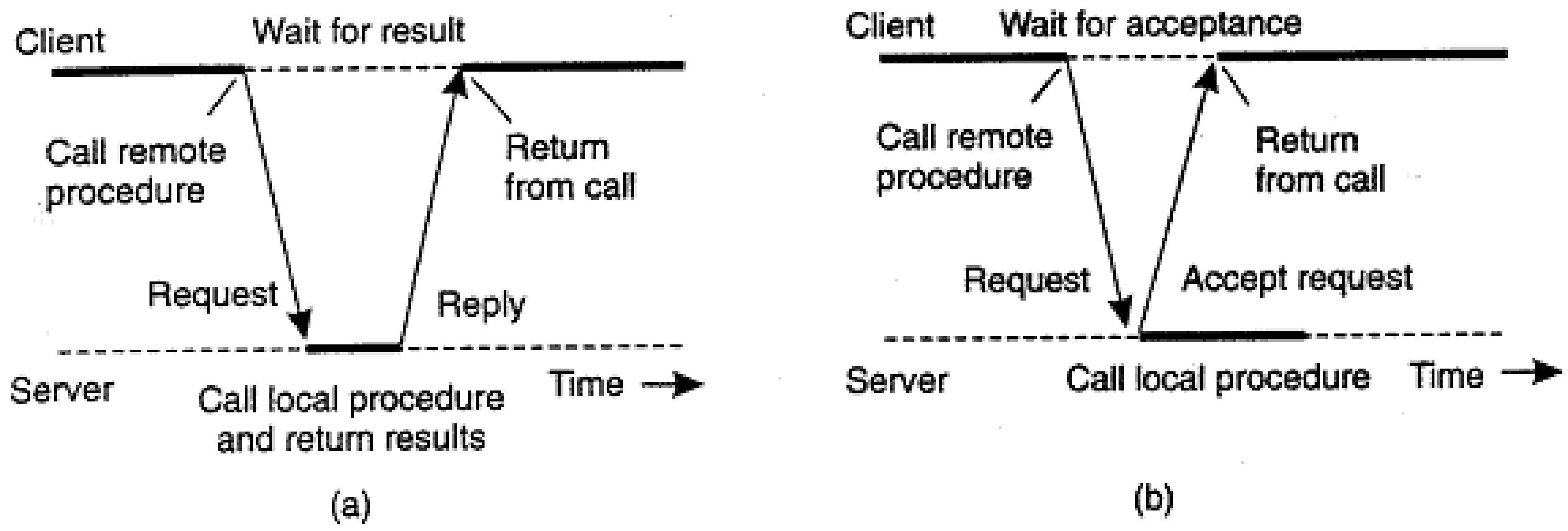


Figure 4-10. (a) The interaction between client and server in a traditional RPc. (b) The interaction using asynchronous RPc.

a. Design issues for RPC

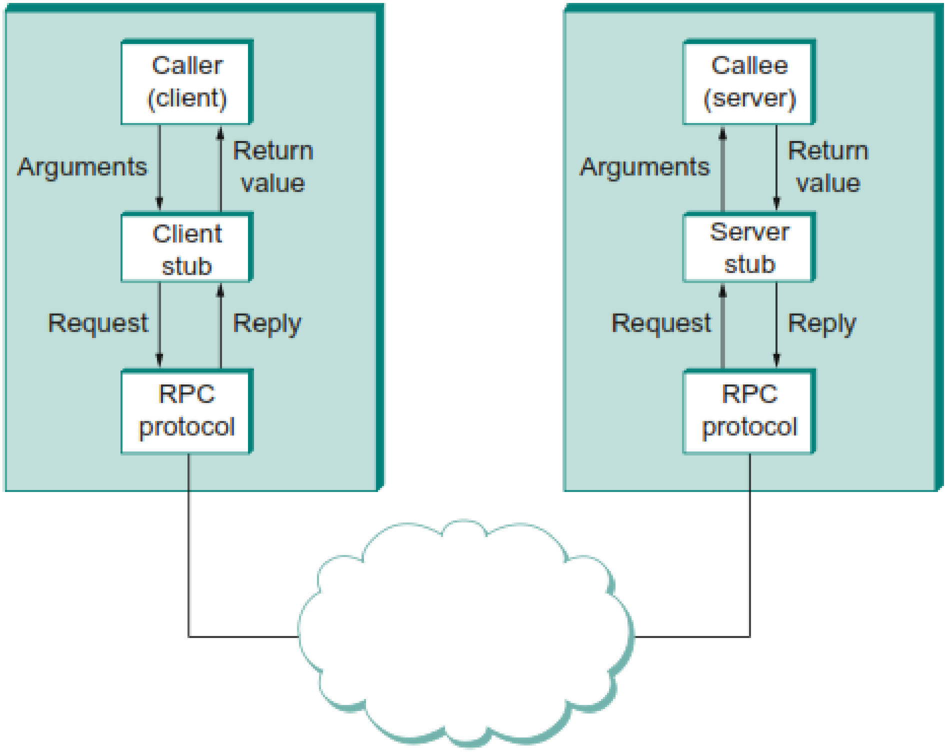
Three issues that are important in understanding this concept:

1. Programming with interfaces :
   a) Interfaces in distributed systems : modular programming, Extrapolating to distributed systems, software evolution
   b) Interface definition languages: allow procedures implemented in different languages to invoke one another
2. RPC call semantics (Retry request message, Duplicate filtering, and Retransmission of results)
   a) Maybe semantics
   b) At-least-once semantics
   c) At-most-once semantics
3. Transparency

Call Semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

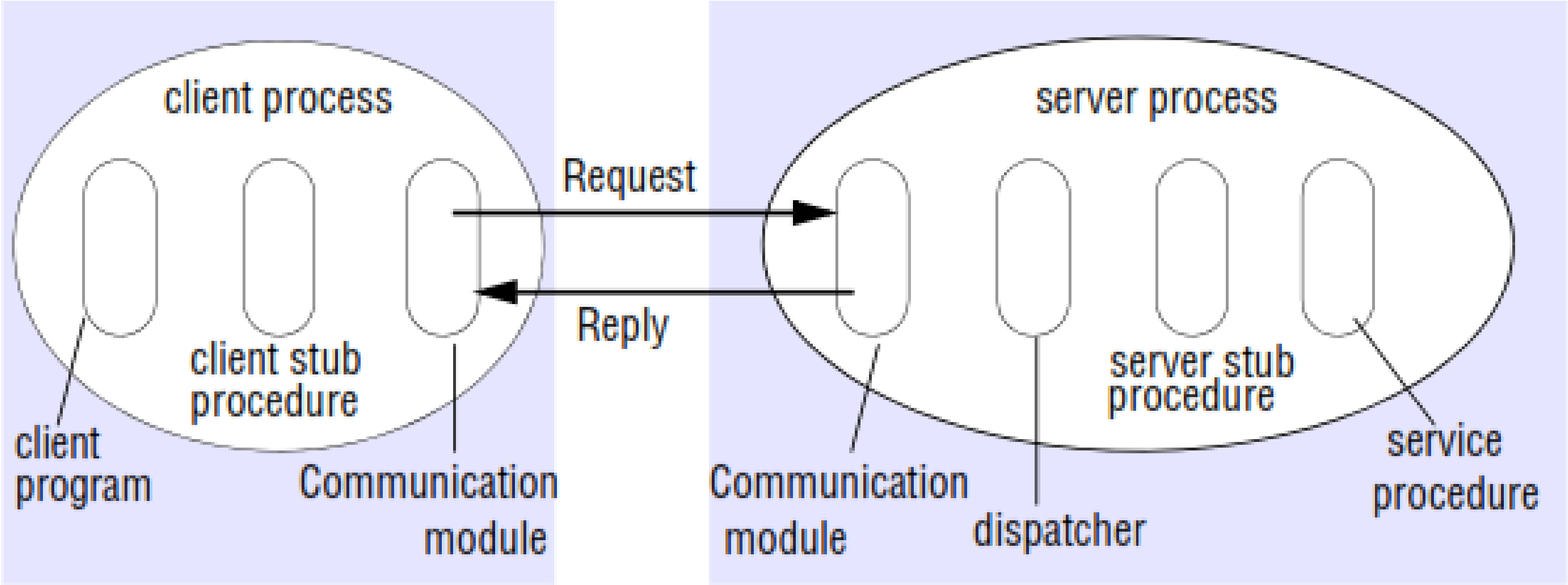
b. Implementation of RPC



We focus on "RPC Protocol (stack)" [**PET'12**]

- ✓ fragments and reassembles large messages (by BLAST)
- ✓ synchronizes request and reply messages (by CHAN)
- ✓ dispatches request to the correct process/procedure (by SELECT)

Role of Client and Server stub Procedures in RPC



c. Case study: Sun RPC

- **RFC 1831** [Srinivasan 1995a] describes Sun RPC, which was designed for client-server communication in the Sun Network File System (NFS).
- Sun RPC is sometimes called ***ONC (Open Network Computing)*** RPC.
- The Sun RPC system provides an interface language called XDR and an interface compiler called rpcgen, which is intended for use with the C programming language.

c. Case study: Sun RPC

Sun RPC details :

- Interface definition language : The Sun XDR language
- Binding : a local binding service called the port mapper (well-known)
- Authentication : request and reply messages
- Client and server programs : www.cdk5.net/rmi

3. Remote Method Invocation

RMI vs RPC

Commonalities	Differences
support programming with interfaces	full expressive power of object-oriented programming in the development of distributed systems software
constructed on top of request-reply protocols and can offer a range of call semantics	all objects in an RMI-based system have unique object references
similar level of transparency	

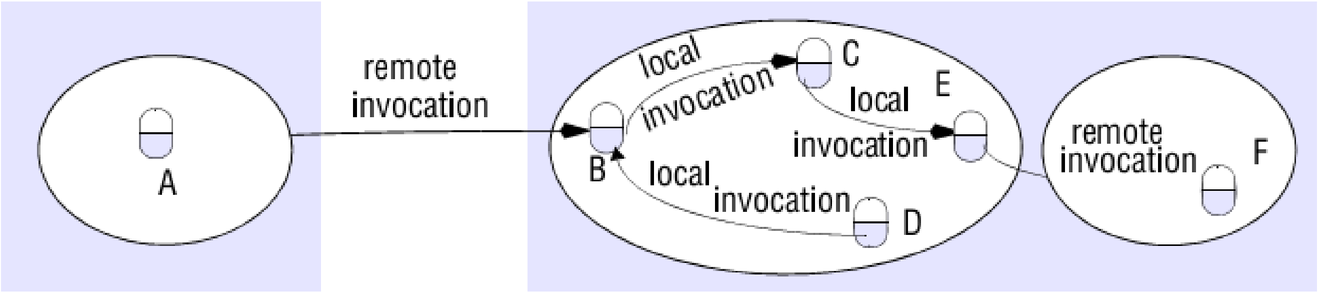
a. Design issues for RMI

The key added design issue relates to the object model and, in particular, achieving the transition from objects to distributed objects.

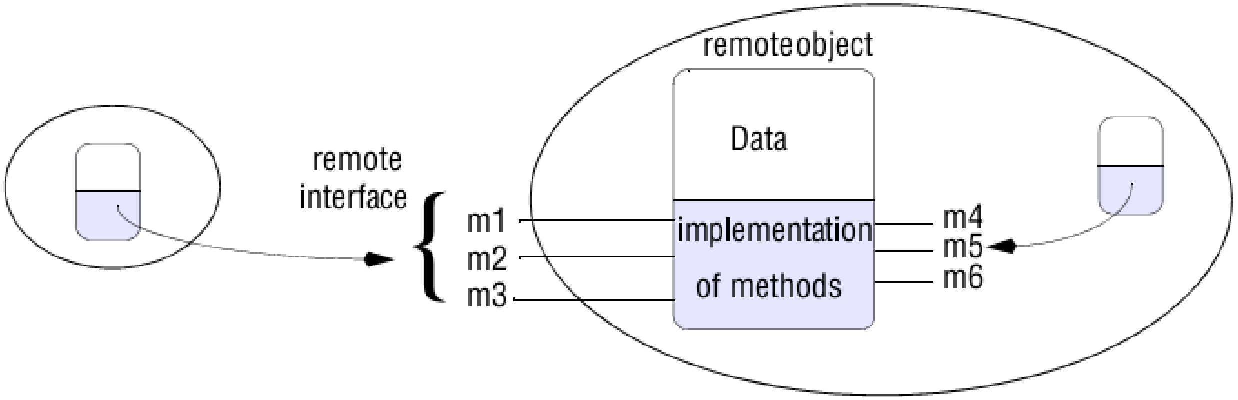
1) The object model : Object references, Interfaces, Actions, Exceptions, and Garbage collection
2) Distributed objects : The state of an object consists of the values of its instance variables
3) The distributed object model : extensions to the object model to make it applicable to distributed objects
4) Actions in a distributed object system : is initiated by a method invocation, which may result in further invocations on methods in other objects.

objects are accessed

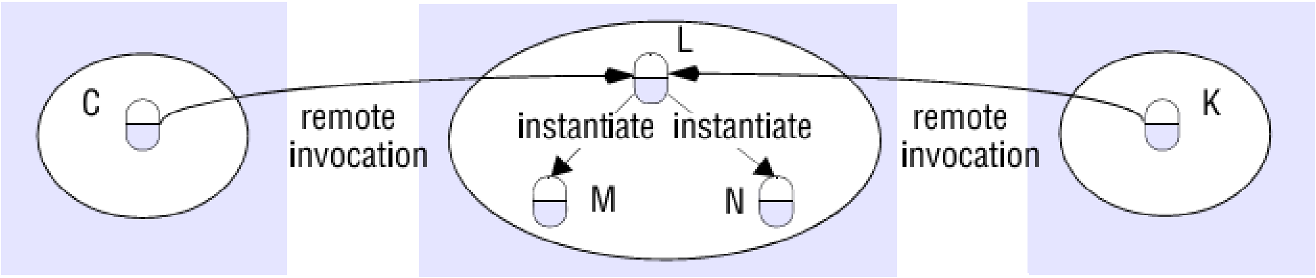
Remote and local method invocations



A remote object and its remote interface



Instantiation of remote objects



b. Implementation of RMI

- Communication module : the message type, its requestId and the remote reference of the object to be invoked.
- Remote reference module : responsible for translating between local and remote object references and for creating remote object references.
- Servants : is an instance of a class that provides the body of a remote object
- The RMI software : Proxy, Dispatcher, and Skeleton.
- Dynamic invocation : An alternative to proxies
- Server and client programs : java.sun.com

c. Distributed Garbage Collection

- Distributed Garbage Collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, the object itself will continue to exist
- The distributed garbage collector works in cooperation with the local garbage collectors :
  a) Each server process maintains a set of the names of the processes
  b) When a client C first receives a remote reference to a particular remote object, B, it makes an addRef(B) invocation to the server
  c) When a client C's garbage collector notices that a proxy for remote object B is no longer reachable, it makes a removeRef(B) invocation to the corresponding server
  d) When B.holders is empty, the server's local garbage collector will reclaim the space occupied by B

d. Case study: Java RMI

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

```
import java.util.Vector;
public class ShapeListServant implements ShapeList {
    private Vector theList;         // contains the list of Shapes
    private int version;
    public ShapeListServant(){...}
    public Shape newShape(GraphicalObject g) {
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }
    public  Vector allShapes(){...}
    public int getVersion() { ... }
}
```

Outline Today

Chapter 5 – Remote Invocation.

Chapter 6 – Indirect Communication.

- Request-Reply Protocols
- RPC
- Remote Method Invocation
- Group Communication
- Shared Memory Approach

Indirect Communication

Indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s).

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 6.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

4. Group Communication

- Group communication provides our first example of an indirect communication paradigm.
- Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group.
- In this action, the sender is not aware of the identities of the receivers.
- Group communication represents an abstraction over multicast communication and may be implemented over IP multicast or an equivalent overlay network, adding significant extra value in terms of managing group membership, detecting failures and providing reliability and ordering guarantees.

a. The programming model

In group communication, the central concept is that of a group with associated group membership, whereby processes may join or leave the group.

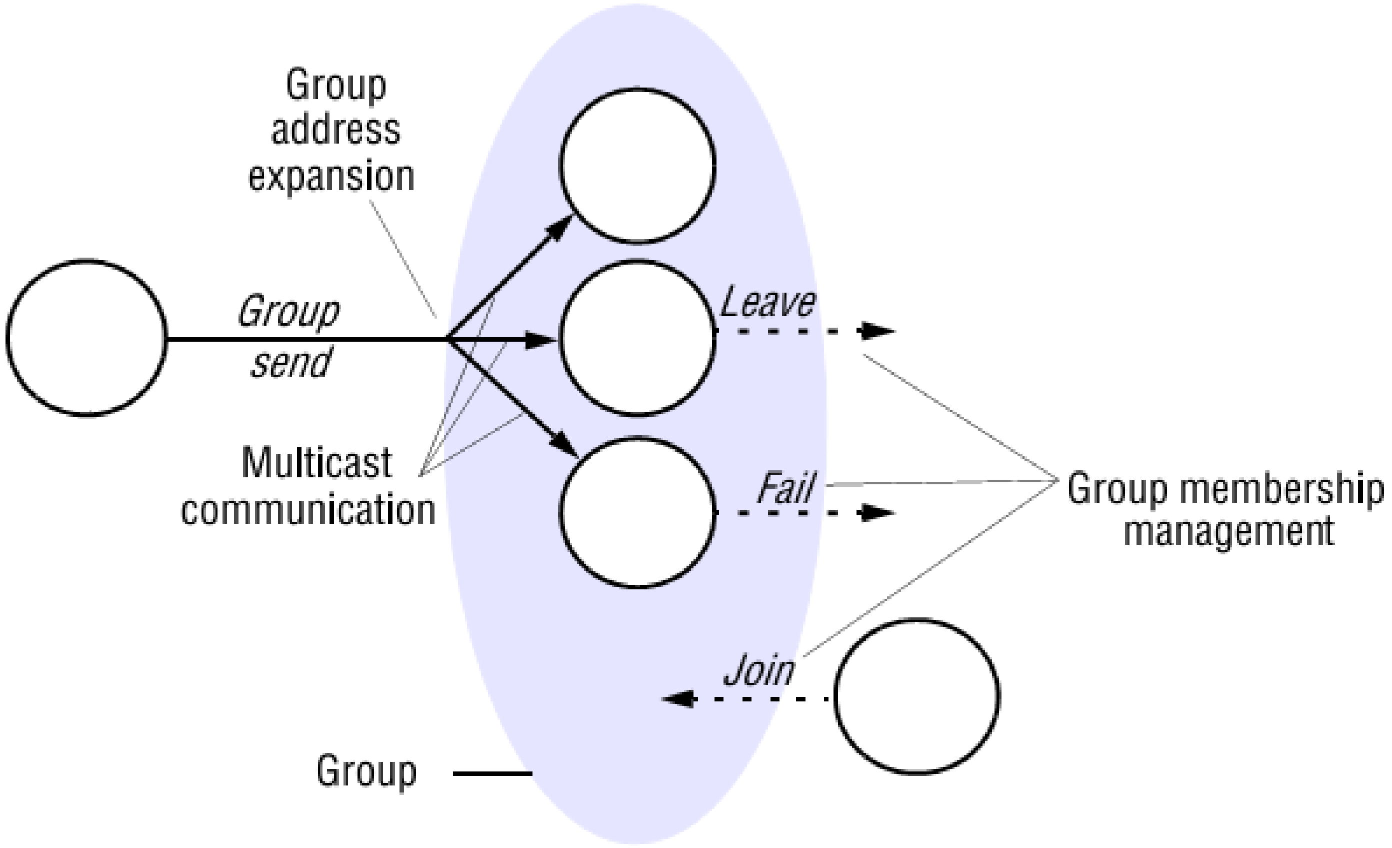
- Process groups and object groups : groups where the communicating entities are processes
- Other key distinctions :Closed and open groups

b. Implementation Issues

The properties of the underlying multicast service in terms of reliability and ordering and also the key role of group membership management in dynamic environments, where processes can join and leave or fail at any time.

- Reliability and ordering in multicast : delivery guarantees and scheduling
- Group membership management : Providing an interface for group membership changes, Failure detection, Notifying members of group membership changes, and Performing group address expansion

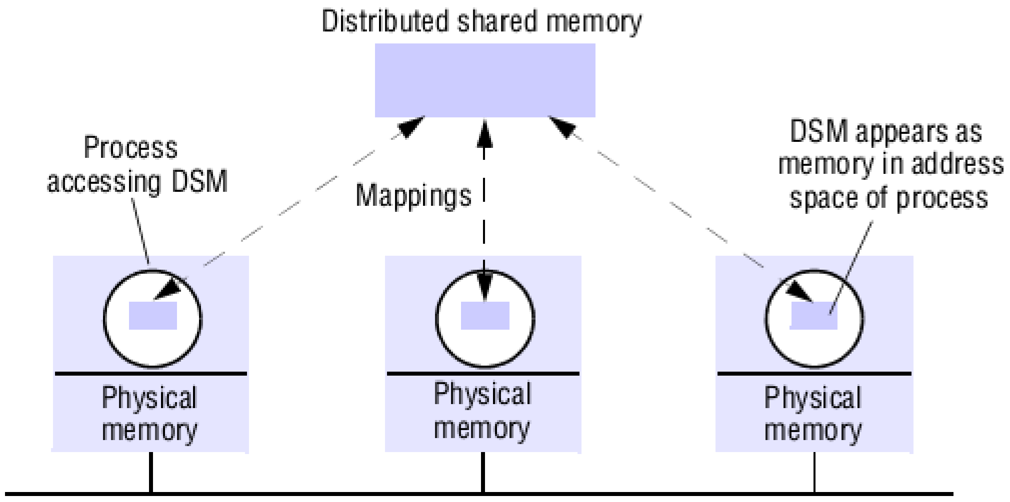
The role of group membership management



5. Shared Memory Approaches

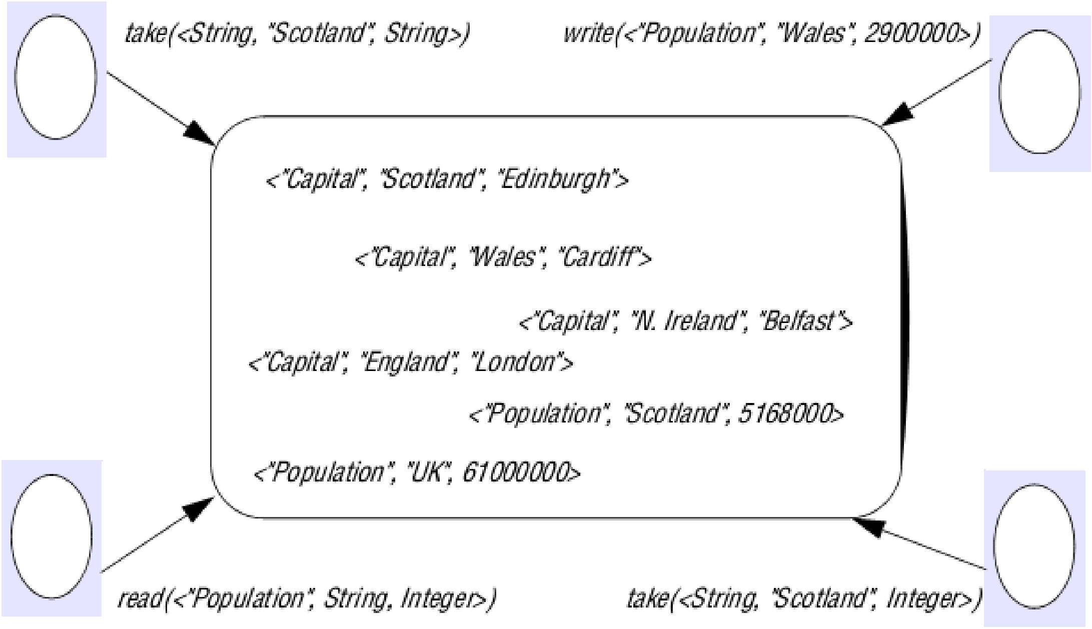
- Distributed shared memory techniques that were developed principally for parallel computing before moving on to tuple space communication, an approach that allows programmers to read and write tuples from a shared tuple space.
- Whereas distributed shared memory operates at the level of reading and writing bytes, tuple spaces offer a higher-level perspective in the form of semi-structured data.

a. Distributed Shared Memory



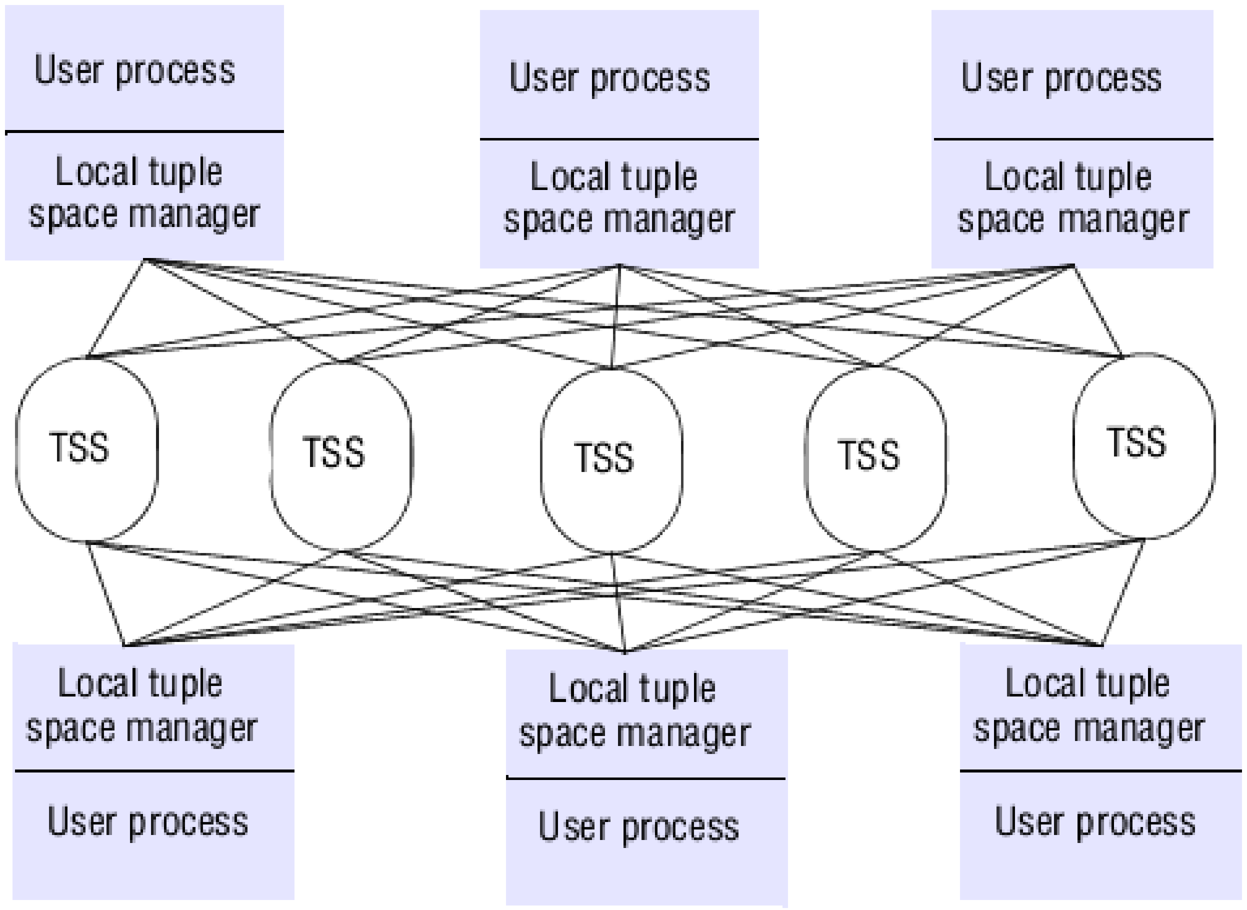
- Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory.
- Processes access DSM by reads and updates to what appears to be ordinary memory within their address space.

b. Tuple Space Communication



- In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them.
- Tuples do not have an address but are accessed by pattern matching on content

Partitioning in the York Linda Kernel



References

[COU’12] Coulouris, G. Dollimore, J., Kindberg, T., Blair, G., DISTRIBUTED SYSTEMS :Concepts and Design Fifth Edition, Pearson Education, Inc., United States of America, 2012.

[TAN’07] Tanenbaum, A.S., Steen, M.V., DISTRIBUTED SYSTEMS : Principles and Paradigms Second Edition, Pearson Education, Inc., United States of America, 2007.

[PET’12] Peterson, L.L., and Davie, B.S., Computer Networks: A Systems Approach Fifth Edition, Morgan Kaufmann, Burlington USA, 2012.

Thank You

Gandeva Bayu Satrya, ST., MT.
Telematics Labz.
Informatics Department
Telkom University
@2014