FACULTY
OF INFORMATICS
Masaryk University

PA160: Net-Centric Computing II.

Distributed Systems

Luděk Matyska

Slides by: Tomáš Rebok

Spring 2019

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

Conclusion

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

Conclusion

Distributed Systems – Definition

Distributed System by Coulouris, Dollimore, and Kindberg

A system in which hardware and software components located at networked computers communicate and coordinate their actions only by message passing.

Distributed System by Tanenbaum and Steen

A collection of independent computers that appears to its users as a single coherent system.

- the independent/autonomous machines are interconnected by communication networks and equipped with software systems designed to produce an integrated and consistent computing environment

Core objective of a distributed system: resource sharing

Distributed Systems – Key characteristics

- **Autonomicity** – there are several autonomous computational entities, each of which has its own local memory
- **Heterogeneity** – the entities may differ in many ways
  - computer HW (different data types' representation), network interconnection, operating systems (different APIs), programming languages (different data structures), implementations by different developers, etc.
- **Concurrency** – concurrent (distributed) program execution and resource access
- **No global clock** – programs (distributed components) coordinate their actions by exchanging messages
  - message communication can be affected by delays, can suffer from variety of failures, and is vulnerable to security attacks
- **Independent failures** – each component of the system can fail independently, leaving the others still running (and possibly not informed about the failure)
  - How to know/differ the states when a network has failed or became unusually slow?
  - How to know immediately if a remote server crashed?

Distributed Systems – Challenges and Issues

What do we want from a Distributed System (DS)?

- *Resource Sharing*
- *Openness*
- *Concurrency*
- *Scalability*
- *Fault Tolerance*
- *Security*
- *Transparency*

Distributed Systems – Challenges and Issues

Resource Sharing

- main motivating factor for constructing DSs
- is should be easy for the users (and applications) to access remote resources, and to share them in a *controlled and efficient way*
  - each resource must be managed by a software that provides interfaces which enable the resource to be manipulated by clients
  - *resource* = anything you can imagine (e.g., storage facilities, data, files, Web pages, etc.)

Distributed Systems – Challenges and Issues

Openness

- whether the system can be extended and re-implemented in various ways and new resource-sharing services can be added and made available for use by a variety of client programs
  - specification and documentation of key software interfaces must be published
    - using an Interface Definition Language (IDL)
- involves HW extensibility as well
  - i.e., the ability to add hardware from different vendors

Distributed Systems – Challenges and Issues

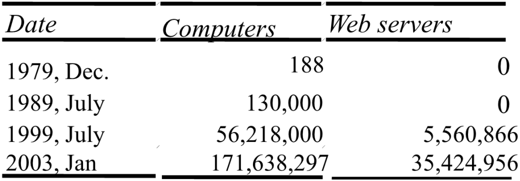
Concurrency

- every resource in a DS must be designed to be safe in a concurrent environment
  - applies not only to servers, but to objects in applications as well
- ensured by standard techniques, like *semaphores*

Distributed Systems – Challenges and Issues

Scalability

- a DS is scalable if the cost of adding a user (or resource) is a constant amount in terms of resources that must be added
  - and is able to utilize the extra hardware/software *efficiently*
  - and remains manageable



Date	*Computers*	*Web servers*
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan	171,638,297	35,424,956

Distributed Systems – Challenges and Issues

Fault Tolerance

- a characteristic where a distributed system provides an appropriately handling of errors that occurred in the system
  - the failures can be *detected* (sometimes hard or even impossible), *masked* (made hidden or less severe), or *tolerated*
- achieved by deploying two approaches: *hardware redundancy* and *software recovery*

Security

- involves *confidentiality, integrity, authentication,* and *availability*

Distributed Systems – Challenges and Issues

Transparency I.

Transparency

- certain aspects of the DS should be made invisible to the user / application programmer
    - i.e., the system is perceived as a whole rather than a collection of independent components
- several *forms of transparency*:
    - **Access transparency** – enables local and remote resources to be accessed using identical operations
    - **Location transparency** – enables resources to be accessed without knowledge of their location
    - **Concurrency transparency** – enables several processes to operate concurrently using shared resources without interference between them
    - **Replication transparency** – enables multiple instances of resources to be used to increase reliability and performance
        - without knowledge of the replicas by users / application programmers

Distributed Systems – Challenges and Issues

Transparency II.

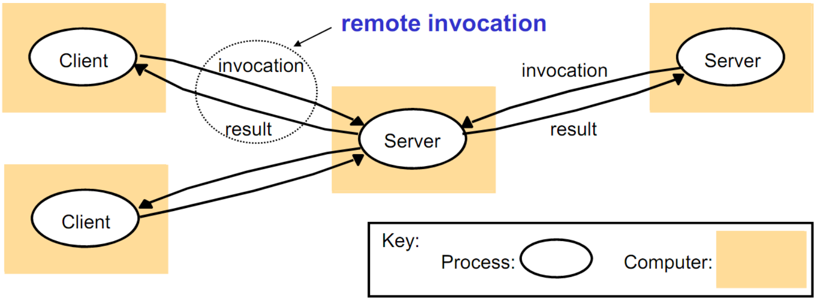
- *forms of transparency* cont'd.:
  - **Failure transparency** – enables the concealment of faults, allowing users and application programs to complete their tasks despite of a failure of HW/SW components
  - **Mobility/migration transparency** – allows the movement of resources and clients within a system without affecting the operation of users or programs
  - **Performance transparency** – allows the system to be reconfigured to improve performance as loads vary
  - **Scaling transparency** – allows the system and applications to expand in scale without changes to the system structure or application algorithms

Distributed Systems – Architecture Models

- defines the way in which the components of systems interact with one another, and
- defines the way in which the components are mapped onto an underlying network of computers
- the overal goal is to ensure that the structure will meet present and possibly future demands
  - the major concerns are to make system reliable, manageable, adaptable, and cost-effective
- principal architecture models:
  - **client-server model** – most important and most widely used
    - a service may be further provided by multiple servers
    - the servers may in turn be clients for another servers
    - proxy servers (caches) may be employed to increase availability and performance
  - **peer processes** – all the processes play similar roles
    - based either on structured (Chord, CAN, etc.), unstructured, or hybrid architectures

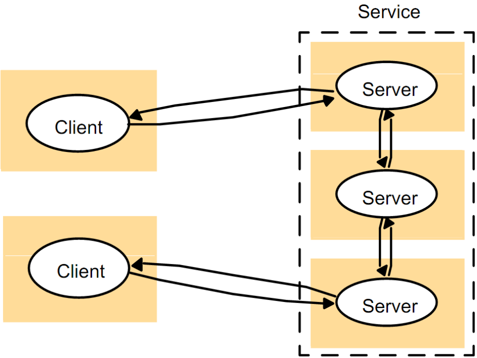
Distributed Systems – Architecture Models

Client-Server model



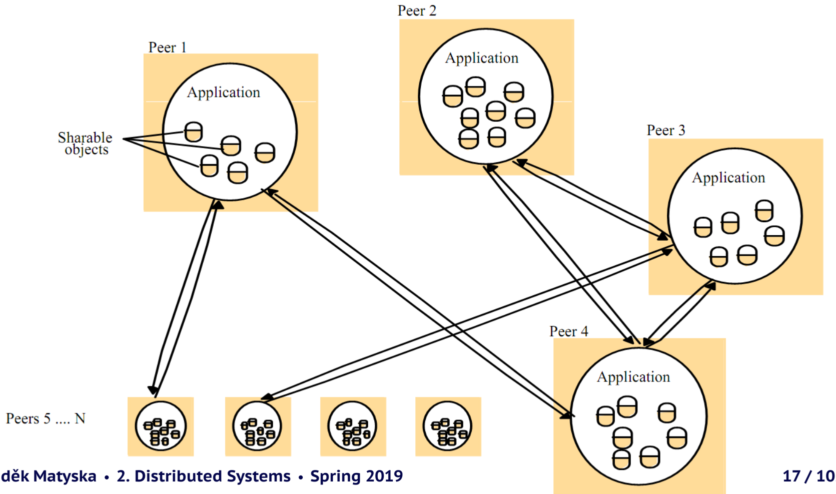
Distributed Systems – Architecture Models

Client-Server model – A Service provided by Multiple Servers



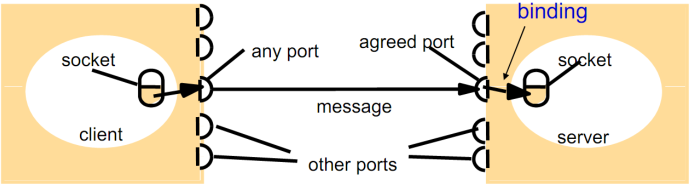
Distributed Systems – Architecture Models

Peer processes



Distributed Systems – Inter-process Communication (IPC)

- the processes (components) need to communicate
- the communication may be:
  - synchronous – both send and receive are blocking operations
  - asynchronous – send is non-blocking and receive can have blocking (more common) and non-blocking variants
- the simplest forms of communication: **UDP** and **TCP sockets**



Distributed Systems – Inter-process Communication

UDP and TCP sockets

UDP/TCP sockets

- provide unreliable/reliable communication services

+ the complete control over the communication lies in the hands of applications

– too primitive to be used in developing a distributed system software
  - higher-level facilities (marshalling/unmarshalling data, error detection, error recovery, etc.) must be built from scratch by developers on top of the existing socket primitive facilities
  - force read/write mechanism instead of a *procedure call*

– another problem arises when the software needs to be used in a platform different from where it was developed
  - the target platform may provide different socket implementation

⇒ *these issues are eliminated by the use of a* ***Middleware***

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

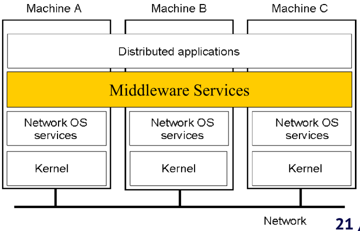
Conclusion

Middleware

Middleware

- a software layer that provides a programming abstraction as well as masks the heterogeneity of the underlying networks, hardware, operating systems, and programming languages
  - ⇒ provides transparency services
  - represented by processes/objects that interact with each other to implement communication and resource sharing support
  - provides building blocks for the construction of SW components that can work with one another

- middleware examples:
  - Sun RPC (ONC RPC)
  - DCE RPC
  - MS COM/DCOM
  - Java RMI
  - CORBA
  - etc.



Middleware

Basic Services

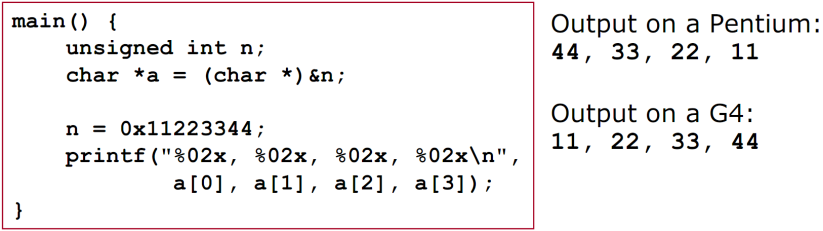
- *Directory services* – services required to locate application services and resources, and route messages
    - $\approx$ service discovery
- *Data encoding services* – uniform data representation services for dealing with incompatibility problems on remote systems
    - e.g., Sun XDR, ISO's ASN.1, CORBA's CDR, XML, etc.
    - data marshalling/unmarshalling
- *Security services* – provide inter-application client-server security mechanisms
- *Time services* – provide a universal format for representing time on different platforms (possibly located in various time zones) in order to keep synchronisation among application processes
- *Transaction services* – provide transaction semantics to support commit, rollback, and recovery mechanisms

Middleware

Basic Services – A Need for Data Encoding Services

Data encoding services are required, because remote machines may have:

- different byte ordering
- different sizes of integers and other types
- different floating point representations
- different character sets
- alignment requirements



```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
            a[0], a[1], a[2], a[3]);
}
```

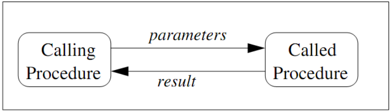
Output on a Pentium:
44, 33, 22, 11

Output on a G4:
11, 22, 33, 44

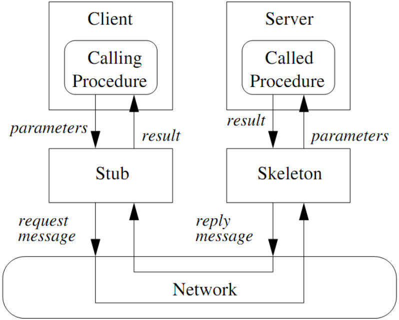
Remote Procedure Calls (RPC)

- very simple idea similar to a well-known procedure call mechanism
    - a client sends a request and blocks until a remote server sends a response
- the goal is to allow distributed programs to be written in the same style as conventional programs for centralised computer systems
    - while being transparent – the programmer need not be aware that the called procedure is executing on a local or a remote computer
- the idea:
    - the remote procedure is represented as a **stub** on the client side
        - behaves like a local procedure, but rather than placing the parameters into registers, it packs them into a message, issues a send primitive, and blocks itself waiting for a reply
    - the server passes the arrived message to a **server stub** (known as **skeleton** as well)
        - the skeleton unpacks the parameters and calls the procedure in a conventional manner
        - the results are returned to the skeleton, which packs them into a message directed to the client stub

Remote Procedure Calls (RPC)



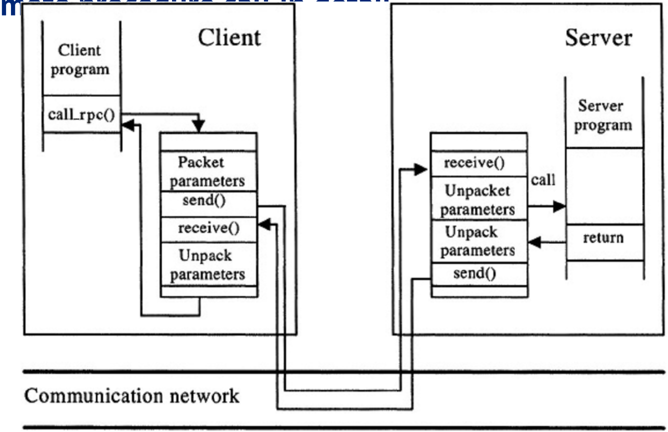
Local Procedure Call



Remote Procedure Call

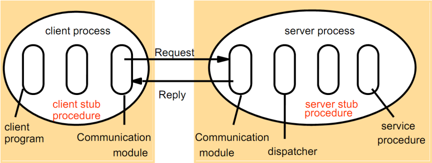
Remote Procedure Calls (RPC)

The remote procedure call in detail



Remote Procedure Calls (RPC)

Components



- client program, client stub
- communication modules
- server stub, service procedure
- dispatcher – selects one of the server stub procedures according to the procedure identifier in the request message

Sun RPC: the procedures are identified by:

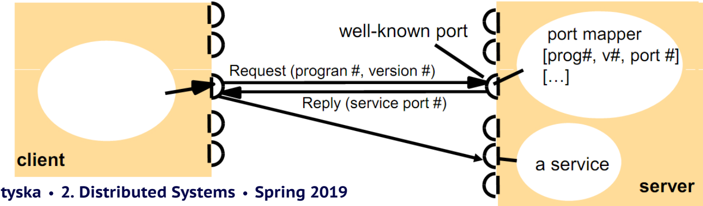
- program number – can be obtained from a central authority to allow every program to have its own unique number
- procedure number – the identifier of the particular procedure within the program
- version number – changes when a procedure signature changes

Remote Procedure Calls (RPC)

Location Services – Portmapper

Clients need to know the port number of a service provided by the server ⇒ **Portmapper**

- a server registers its *program#*, *version#*, and *port#* to the local portmapper
- a client finds out the *port#* by sending a request
  - the portmapper listens on a *well-known port* (111)
  - the particular procedure required is identified in the subsequent procedure call



Remote Procedure Calls (RPC)

Parameters passing

How to pass parameters to remote procedures?

- **pass by value** – easy: just copy data to the network message
- **pass by reference** – makes no sense without shared memory

Pass by reference: the steps

1. copy referenced items (marshalled) to a message buffer
2. ship them over, unmarshal data at server
3. pass local pointer to server stub function
4. send new values back

- *to support complex structures:*
  - copy the structure into pointerless representation
  - transmit
  - reconstruct the structure with local pointers on the server

Remote Procedure Calls (RPC)

Parameters passing – eXternal Data Representation (XDR)

Sun RPC: to avoid compatibility problems, the **eXternal Data Representation (XDR)** is used

- XDR primitive functions examples:
  - `xdr_int()`, `xdr_char()`, `xdr_u_short()`, `xdr_bool()`, `xdr_long()`, `xdr_u_int()`, `xdr_wrapstring()`, `xdr_short()`, `xdr_enum()`, `xdr_void()`
- XDR aggregation functions:
  - `xdr_array()`, `xdr_string()`, `xdr_union()`, `xdr_vector()`, `xdr_opaque()`
- *only a single input parameter is allowed in a procedure call*
  - ⇒ procedures requiring multiple parameters must include them as components of a single structure

Remote Procedure Calls (RPC)

When Things Go Wrong I.

- local procedure calls do not fail
    - if they core dump, entire process dies
- there are more opportunities for errors with RPC
    - server could generate an error
    - problems in network (lost/delayed requests/replies)
    - server crash
    - client might crash while server is still executing code for it
- transparency breaks here
    - applications should be prepared to deal with RPC failures

Remote Procedure Calls (RPC)

When Things Go Wrong II.

Semantics of local procedure calls: **exactly once**

- difficult to achieve with RPC

Four remote calls semantics available in RPC:

- **at-least-once semantic**
  - client keeps trying sending the message until a reply has been received
    - failure is assumed after *n* re-sends
  - guarantees that the call has been made “at least once”, but possibly multiple times
  - ideal for idempotent operations
- **at-most-once semantic**
  - client gives up immediately and reports back a failure
  - guarantees that the call has been made “at most once”, but possibly none at all
- **exactly-once semantic**
  - the most desirable, but the most difficult to implement
- **maybe semantic**

Remote Procedure Calls (RPC)

When Things Go Wrong III.

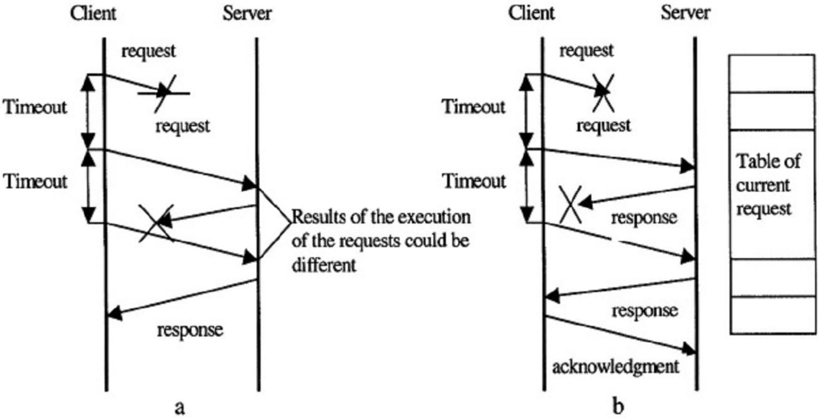
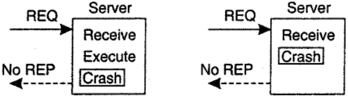


Figure: Message-passing semantics. (a) at-least-once; (b) exactly-once.

Remote Procedure Calls (RPC)

When Things Go Wrong IV. – Complications

1. it is necessary to understand the application
   - idempotent functions – in the case of a failure, the message may be retransmitted and re-run without a harm
   - non-idempotent functions – has side-effects ⇒ the retransmission has to be controlled by the server
     - the duplicity request (retransmission) has to be detected
     - once detected, the server procedure is NOT re-run; just the results are resent (if available in a server cache)
2. in the case of a server crash, the order of execution vs. crash matters



3. in the case of a client crash, the procedure keeps running on the server
   - consumes resources (e.g., CPU time), possesess resources (e.g., locked files), etc.
   - may be overcome by employing soft-state principles
     - keep-alive messages

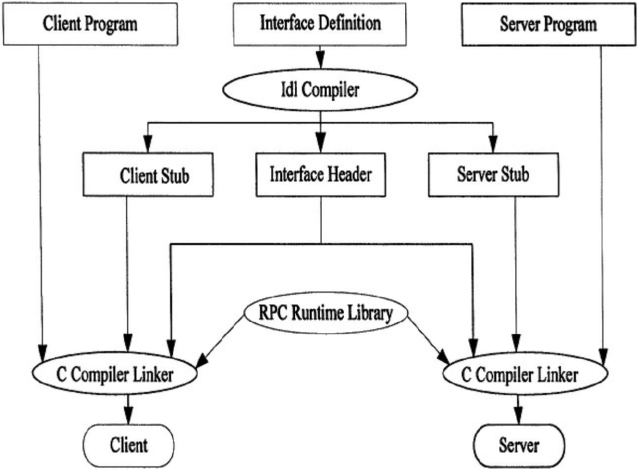
Remote Procedure Calls (RPC)

Code Generation I.

- RPC drawbacks:
  - complex API, not easy to debug
  - the use of XDR is difficult
- *but, it's often used in a similar way*
- ⇒ the server/client code can be automatically generated
  - assumes well-defined interfaces (IDL)
  - the application programmer has to supply the following:
    - interface definition file – defines the interfaces (data structures, procedure names, and parameters) of the remote procedures that are offered by the server
    - client program – defines the user interfaces, the calls to the remote procedures of the server, and the client side processing functions
    - server program – implements the calls offered by the server
  - compilers:
    - `rpcgen` for C/C++, `jrpcgen` for Java

Remote Procedure Calls (RPC)

Code Generation II.



Remote Method Invocation (RMI)

- as the popularity of object technology increased, techniques were developed to allow calls to remote objects instead of remote procedures only

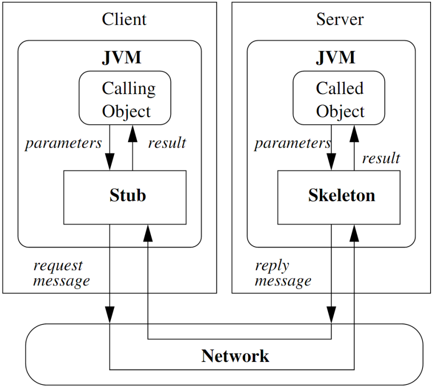
⇒ **Remote Method Invocation (RMI)**

- essentially the same as the RPC, except that it operates on objects instead of applications/procedures
  - the RMI model represents a distributed object application
    - it allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client
  - the server application creates an object and makes it accesible remotely (i.e., registers it)
  - the client application receives a reference to the object on the server and invokes methods on it
    - the reference is obtained through looking up in the registry
  - **important:** a method invocation on a remote object has the same syntax as a method invocation on a local object

Remote Method Invocation (RMI)

Architecture I.

The interface, through which the client and server interact, is (similarly to RPC) provided by **stubs** and **skeletons**:

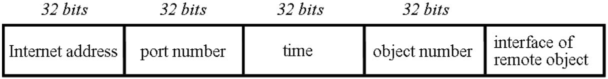


Remote Method Invocation (RMI)

Architecture II.

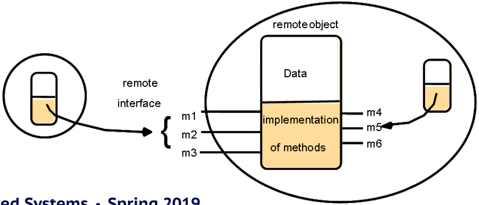
Two fundamental concepts as the heart of distributed object model:

- *remote object reference* – an identifier that can be used throughout a distributed system to refer to a particular unique remote object
  - its construction must ensure its uniqueness



32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

- *remote interface* – specifies, which methods of the particular object can be invoked remotely



Remote Method Invocation (RMI)

Architecture III.

- the remote objects can be accessed concurrently
  - the encapsulation allows objects to provide methods for protecting themselves against incorrect accesses
    - e.g., synchronization primitives (condition variables, semaphores, etc.)
- RMI transaction semantics similar to the RPC ones
  - at-least-once, at-most-once, exactly-once, and maybe semantics
- data encoding services:
  - stubs use Object Serialization to marshal the arguments
    - object arguments' values are rendered into a stream of bytes that can be transmitted over a network
    - ⇒ the arguments must be primitive types or objects that implement `Serializable` interface
- parameters passing:
  - local objects passed by value
  - remote objects passed by reference

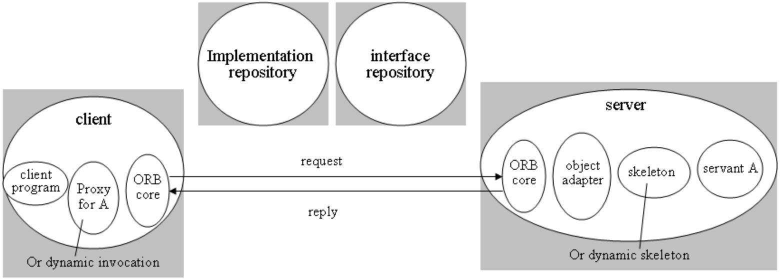
Common Object Request Broker Architecture (CORBA)

- an industry standard developed by the OMG (*Object Management Group* – a consortium of more than 700 companies) to aid in distributed objects programming
  - OMG was established in 1988
  - initial CORBA specification came out in 1992
    - but significant revisions have taken place from that time
- provides a **platform-independent** and **language-independent** architecture (framework) for writing distributed, object-oriented applications
  - i.e., application programs can communicate without restrictions to:
    - programming languages, hardware platforms, software platforms, networks they communicate over
  - but CORBA is just a specification for creating and using distributed objects; it is not a piece of software or a programming language
    - several implementations of the CORBA standard exist (e.g., IBM's SOM and DSOM architectures)

CORBA Components

CORBA is composed of five major components:

- *Object Request Broker (ORB)*
- *Interface Definition Language (IDL)*
- *Dynamic Invocation Interface (DII)*
- *Interface Repositories (IR)*
- *Object Adapters (OA)*



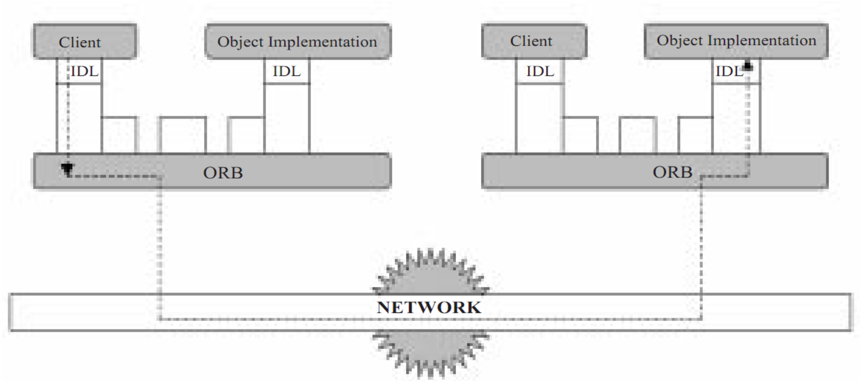
CORBA Components

Object Request Broker (ORB)

- the heart of CORBA
  - introduced as a part of OMG's Object Management Architecture (OMA), which the CORBA is based on
- a distributed service that implements all the requests to the remote object(s)
  - it **locates the remote object** on the network, **communicates the request** to the object, waits for the results and (when available) **communicates those results** back to the client
- implements location transparency
  - exactly the same request mechanism is used regardless of where the object is located
    - might be in the same process with the client or across the planet
- implements programming language independence
  - the client issuing a request can be written in a different programming language from the implementation of the CORBA object
- both the client and the object implementation are isolated from the ORB by an IDL interface
- Internet Inter-ORB Protocol (IIOP) – the standard communication protocol between ORBs

CORBA Components

Object Request Broker (ORB) II.



CORBA Components

Interface Definition Language (IDL) I.

Interface Definition Language (IDL)

- as with RMI, CORBA objects have to be specified with interfaces
  - interface ≈ a contract between the client (code using a object) and the server (code implementing the object)
    - indicates a set of operations the object supports and how they should be invoked (but NOT how they are implemented)
- defines modules, interfaces, types, attributes, exceptions, and method signatures
  - uses same lexical rules as C++
    - with additional keywords to support distribution (e.g. `interface`, `any`, `attribute`, `in`, `out`, `inout`, `readonly`, `raises`)
- defines language bindings for many different programming languages (e.g., C/C++, Java, etc.)
  - via language mappings, the IDL translates to different constructs in the different implementation languages
  - it allows an object implementor to choose the appropriate

CORBA Components

Interface Definition Language (IDL) II.

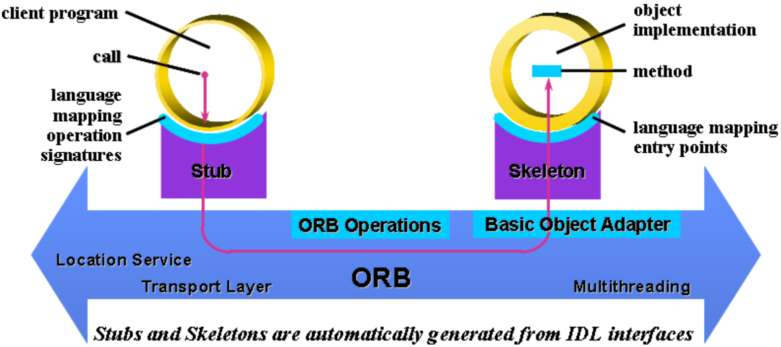
Interface Definition Language (IDL) example:

```
module StockObjects {
    struct Quote {
        string symbol;
        long at_time;
        double price;
        long volume;
    };
    exception Unknown{};
    interface Stock {
        // Returns the current stock quote.
        Quote get_quote() raises(Unknown);
        // Sets the current stock quote.
        void set_quote(in Quote stock_quote);
        // // Provides the stock description, e.g. company name.
        readonly attribute string description;
    };
    interface StockFactory {
        Stock create_stock(in string symbol, in string description);
    };
```

CORBA Components

Interface Definition Language (IDL) III. – Stubs and Skeletons

IDL compiler automatically compiles the IDL into **client stubs** and **object skeletons**:

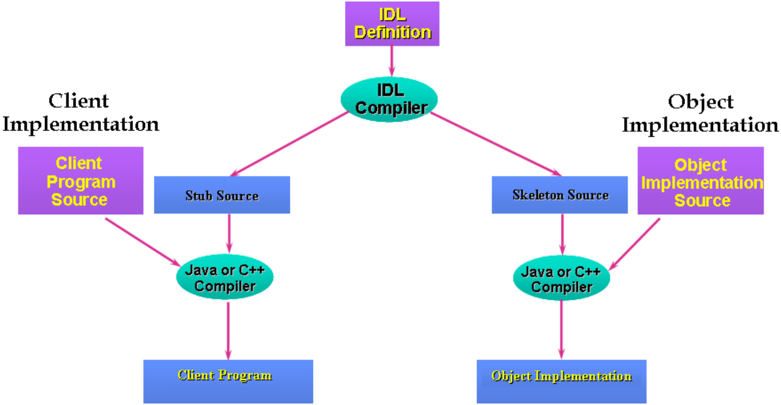


Stubs and Skeletons are automatically generated from IDL interfaces

CORBA Components

Interface Definition Language (IDL) IV. – Development Process

Using IDL



CORBA Components

Dynamic Invocation Interface (DII) & Dynamic Skeleton Interface (DSI)

Dynamic Invocation Interface (DII)

- CORBA supports both the *dynamic* and the *static invocation interfaces*
  - *static invocation interfaces* are determined at compile time
  - *dynamic interfaces* allow client applications to use server objects without knowing the type of those objects at compile time
- DII – an API which allows dynamic construction of CORBA object invocations

Dynamic Skeleton Interface (DSI)

- DSI is the server side's analogue to the client side's DII
  - allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing

CORBA Components

Interface Repository (IR)

Interface Repository (IR)

- a runtime component used to dynamically obtain information on IDL types (e.g. object interfaces)
  - using the IR, a client should be able to locate an object that is unknown at compile time, find information about its interface, and build a request to be forwarded through the ORB
  - this kind of information is necessary when a client wants to use the DII to construct requests dynamically

CORBA Components

Object Adapters (OAs)

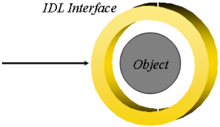
Object Adapters (OAs)

- the interface between the ORB and the server process
  - OAs listen for client connections/requests and map the inbound requests to the desired target object instance
- provide an API that object implementations use for:
  - generation and interpretation of object references
  - method invocation
  - security of interactions
  - object and implementation activation and deactivation
  - mapping object references to the corresponding object implementations
  - registration of implementations
- two basic kinds of OAs:
  - *basic object adapter (BOA)* – leaves many features unsupported, requiring proprietary extensions
  - *portable object adapter (POA)* – intended to support multiple ORB

CORBA Object & Object Reference

CORBA Objects are fully encapsulated

- accessed through well-defined interfaces only
- interfaces & implementations are totally separate
  - for one interface, multiple implementations possible
  - one implementation may be supporting multiple interfaces



CORBA Object Reference is the distributed computing equivalent of a pointer

- CORBA defines the Interoperable Object Reference (IOR)
- an IOR contains a fixed object key, containing:
  - the object's fully qualified interface name (repository ID)
  - user-defined data for the instance identifier
  - can also contain transient information:
    - the host and port of its server, metadata about the server's ORB

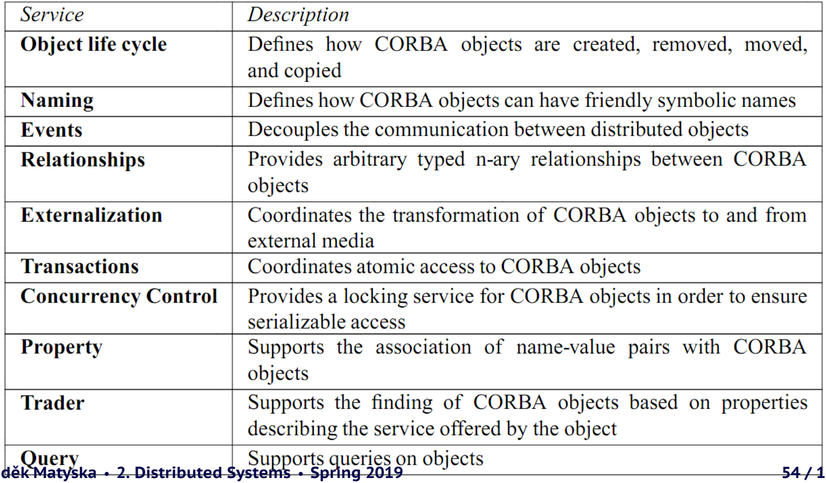
CORBA Services

CORBA Services (COS)

- the OMG has defined a *set of Common Object Services* to support the integration and interoperation of distributed objects
  - = frequently used components needed for building robust applications
  - typically supplied by vendors
  - OMG defines interfaces to services to ensure interoperability

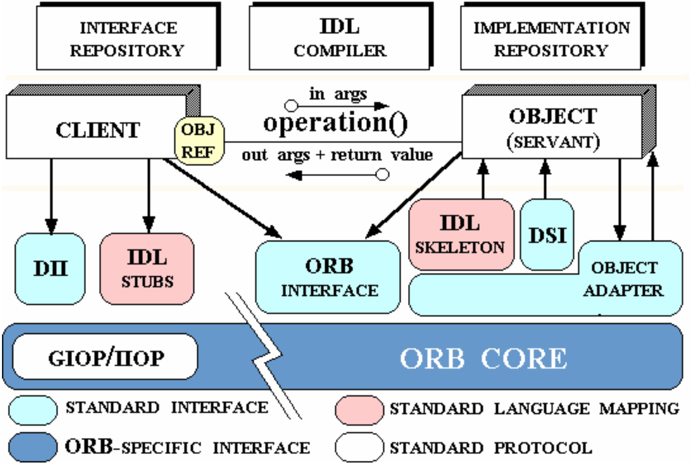
CORBA Services

Popular Services Example



Service	*Description*
Object life cycle	Defines how CORBA objects are created, removed, moved, and copied
Naming	Defines how CORBA objects can have friendly symbolic names
Events	Decouples the communication between distributed objects
Relationships	Provides arbitrary typed n-ary relationships between CORBA objects
Externalization	Coordinates the transformation of CORBA objects to and from external media
Transactions	Coordinates atomic access to CORBA objects
Concurrency Control	Provides a locking service for CORBA objects in order to ensure serializable access
Property	Supports the association of name-value pairs with CORBA objects
Trader	Supports the finding of CORBA objects based on properties describing the service offered by the object
Query	Supports queries on objects

CORBA Architecture Summary



FACULTY
OF INFORMATICS
Masaryk University

Lecture overview

Distributed Systems

Key characteristics

Challenges and Issues

Distributed System Architectures

Inter-process Communication

Middleware

Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

Scheduling/Load-balancing in Distributed Systems

Fault Tolerance in Distributed Systems

Conclusion

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

Conclusion

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

Conclusion

Scheduling/Load-balancing in Distributed Systems I.

For concurrent execution of interacting processes:

- **communication** and **synchronization between processes** are the two essential system components

Before the processes can execute, they need to be:

- **scheduled** and
- allocated with resources

Why scheduling in distributed systems is of special interest?

- because of the issues that are different from those in traditional multiprocessor systems:
  - *the communication overhead is significant*
  - *the effect of underlying architecture cannot be ignored*
  - *the dynamic behaviour of the system must be addressed*
- local scheduling (on each node) + global scheduling

Scheduling/Load-balancing in Distributed Systems II.

Scheduling/Load-balancing in Distributed Systems

- let’s have a pool of jobs
    - there are some inter-dependencies among them
- let’s have a set of nodes (processors), which are able to reciprocally communicate

Load-balancing

The term **load-balancing** means assigning the jobs to the processors in the way, which *minimizes the time/communication overhead necessary to compute them.*

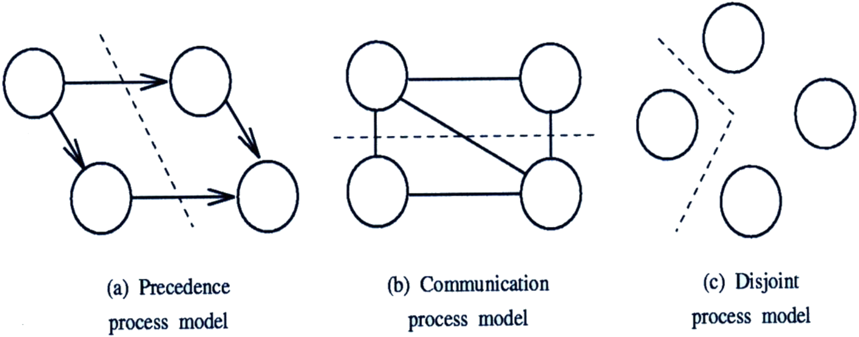
- *load-balancing* – divides the jobs among the processors
- *scheduling* – defines, in which order the jobs have to be executed (on each processor)
    - load-balancing and planning are tightly-coupled (usually considered as synonyms in DSs)
- **objectives:**
    - enhance overall system performance metric

Scheduling/Load-balancing in Distributed Systems III.

- the scheduling/load-balancing task can be represented using graph theory:
    - the pool of N jobs with dependencies can be described as a graph $G(V, U)$, where
        - the nodes represent the jobs (processes)
        - the edges represent the dependencies among the jobs/processes (e.g., an edge from i to j requires that the process i has to complete before j can start executing)
- the graph G has to be splitted into p parts, so that:
    - $N = N_1 \cup N_2 \cup \cdots \cup N_p$
    - which satisfy the condition, that $|N_i| \approx \frac{|N|}{p}$, where
        - $|N_i|$ is the number of jobs assigned to the processor i, and
        - p is the number of processors, and
        - the number/cost of the edges connecting the parts is minimal
    - the *objectives*:

Scheduling/Load-balancing in Distributed Systems III.

An illustration


(a) Precedence process model

(b) Communication process model

(c) Disjoint process model

Figure: An illustration of splitting 4 jobs onto 2 processors.

Scheduling/Load-balancing in Distributed Systems IV.

- the “proper” approach to the scheduling/load-balancing problem depends on the following criteria:
  - *jobs’ cost*
  - *dependencies among the jobs*
  - *jobs’ locality*

Scheduling/Load-balancing in Distributed Systems IV.

Jobs' Cost

- the job's cost may be known:
  - before the whole problem set's execution
  - during problem's execution, but before the particular job's execution
  - just after the particular job finishes
- *cost's variability* – all the jobs may have (more or less) the same cost or the costs may differ
- the problem classes based on jobs' cost:
  - all the jobs have the same cost: *easy*
  - the costs are variable, but, known: *more complex*
  - the costs are unknown in advance: *the most complex*

Scheduling/Load-balancing in Distributed Systems IV.

Dependencies Among the Jobs

- is the order of jobs' execution important?
- the dependencies among the jobs may be known:
  - before the whole problem set's execution
  - during problem's execution, but before the particular job's execution
  - are fully dynamic
- the problem classes based on jobs' dependencies:
  - the jobs are fully independent on each other: *easy*
  - the dependencies are known or predictable: *more complex*
    - flooding
    - in-trees, out-trees (balanced or unbalanced)
    - generic oriented trees (DAG)
  - the dependencies dynamically change: *the most complex*
    - e.g., searching/lookup problems

Scheduling/Load-balancing in Distributed Systems IV.

Locality

- communicate all the jobs in the same/similar way?
- is it suitable/necessary to execute some jobs “close” to each other?
- when the job’s communication dependencies are known?
- the problem classes based on jobs’ locality:
  - the jobs do not communicate (at most during initialization): *easy*
  - the communications are known/predictable: *more complex*
    - regular (e.g., a grid) or irregular
  - the communications are unknown in advance: *the most complex*
    - e.g., a discrete events’ simulation

Scheduling/Load-balancing in DSs – Solving Methods

- in general, the “proper” solving method depends on the time, when the particular information is known
- basic solving algorithms’ classes:
  - *static* – offline algorithms
  - *semi-static* – hybrid approaches
  - *dynamic* – online algorithms
- some (but not all) variants:
  - static load-balancing
  - semi-static load-balancing
  - self-scheduling
  - distributed queues
  - DAG planning

Scheduling/Load-balancing in DSs – Solving Methods

Semi-static load-balancing

Semi-static load-balancing

- suitable for problem sets with slow changes in parameters, and with locality importance
- iterative approach
  - uses static algorithm
  - the result (from the static algorithm) is used for several steps (slight unbalance is accepted)
  - after the steps, the problem set is recalculated with the static algorithm again
- often used for:
  - particle simulation
  - calculations of slowly-changing grids (but in a different sense than in the previous lectures)

Scheduling/Load-balancing in DSs – Solving Methods

Self-scheduling I.

Self-scheduling

- a centralized pool of jobs
- idle processors pick the jobs from the pool
- new (sub)jobs are added to the pool
- + ease of implementation
- suitable for:
    - a set of independent jobs
    - jobs with unknown costs
    - jobs where locality does not matter
- unsuitable for too small jobs – due to the communication overhead
    - ⇒ coupling jobs into bulks
        - fixed size
        - controlled coupling

Scheduling/Load-balancing in DSs – Solving Methods

Self-scheduling II. – Fixed size & Controlled coupling

Fixed size

- typical offline algorithm
- requires much information (number and cost of each job, ...)
- it is possible to find the optimal solution
- theoretically important, not suitable for practical solutions

Controlled coupling

- uses bigger bulks in the beginning of the execution, smaller bulks in the end of the execution
    - lower overhead in the beginning, finer coupling in the end
- the bulk's size is computed as: $K_i = \lceil \frac{R_i}{p} \rceil$
  where:
    - R_i ...the number of remaining jobs
    - p ...the number of processors

Scheduling/Load-balancing in DSs – Solving Methods

Self-scheduling II. – Tapering & Weighted distribution

Tapering

- analogical to the Controlled coupling, but the bulks' size is further a function of jobs' variation
- uses historical information
  - low variance ⇒ bigger bulks
  - high variance ⇒ smaller bulks

Weighted distribution

- considers the nodes' computational power
- suitable for heterogenous systems
- uses historical information as well

Scheduling/Load-balancing in DSs – Solving Methods

Distributed Queues

Distributed Queues

- $\approx$ self-scheduling for distributed memory
- instead of a centralized pool, a queue on each node is used (per-processor queues)
- suitable for:
  - distributed systems, where the locality does not matter
  - for both static and dynamic dependencies
  - for unknown costs
- an example: diffuse approach
  - in every step, the cost of jobs remaining on each processor is computed
  - processors exchange this information and perform the balancing
  - locality must not be important

Scheduling/Load-balancing in DSs – Solving Methods

Centralised Pool vs. Distributed Queues

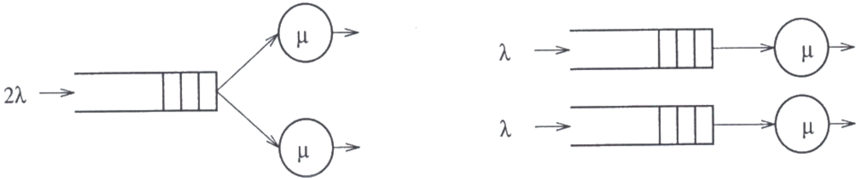


Figure: Centralised Pool (left) vs. Distributed Queues (right).

Scheduling/Load-balancing in DSs – Solving Methods

DAG Planning

DAG Planning

- another graph model
  - the nodes represent the jobs (possibly weighted)
  - the edges represent the dependencies and/or the communication (may be also weighted)
- e.g., suitable for digital signal processing
- basic strategy – divide the DAG so that the communication and the processors' occupation (time) is minimized
  - NP-complete problem
  - takes the dependencies among the jobs into account

Scheduling/Load-balancing in DSs – Design Issues I.

When the scheduling/load-balancing is necessary?

- for middle-loaded systems
  - lowly-loaded systems – rarely job waiting (there's always an idle processor)
  - highly-loaded systems – little benefit (the load-balancing cannot help)

What is the performance metric?

- mean response time

What is the measure of load?

- must be easy to measure
- must reflect performance improvement
- example: queue lengths at CPU, CPU utilization

Scheduling/Load-balancing in DSs – Design Issues I.

Components

Types of policies:

- *static* (decisions hardwired into system), *dynamic* (uses load information), *adaptive* (policy varies according to load)

Policies:

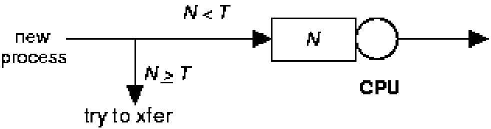
- *Transfer policy:* when to transfer a process?
  - threshold-based policies are common and easy
- *Selection policy:* which process to transfer?
  - prefer new processes
  - transfer cost should be small compared to execution cost
    - ⇒ select processes with long execution times
- *Location policy:* where to transfer the process?
  - polling, random, nearest neighbor, etc.
- *Information policy:* when and from where?
  - demand driven (only a sender/receiver may ask for), time-driven (periodic), state-change-driven (send update if load changes)

Scheduling/Load-balancing in DSs – Design Issues II.

Sender-initiated Policy

Sender-initiated Policy

- *Transfer policy*



- *Selection policy:* newly arrived process
- *Location policy:* three variations
  - Random – may generate lots of transfers
    - ⇒ necessary to limit max transfers
  - Threshold – probe n nodes sequentially
    - transfer to the first node below the threshold, if none, keep job
  - Shortest – poll N_p nodes in parallel
    - choose least loaded node below T

Scheduling/Load-balancing in DSs – Design Issues II.

Receiver-initiated Policy

Receiver-initiated Policy

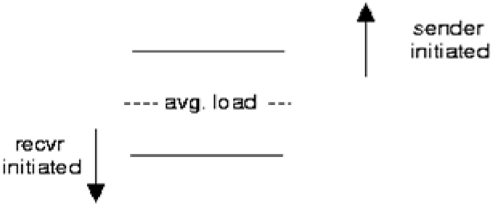
- *Transfer policy:* if departing process causes load $< T$, find a process from elsewhere
- *Selection policy:* newly arrived or partially executed process
- *Location policy:*
  - Threshold – probe up to N_p other nodes sequentially
    - transfer from first one above the threshold; if none, do nothing
  - Shortest – poll n nodes in parallel
    - choose the node with heaviest load above T

Scheduling/Load-balancing in DSs – Design Issues II.

Symmetric Policy

Symmetric Policy

- combines previous two policies without change
    - nodes act as both senders and receivers
- uses average load as the threshold



Scheduling/Load-balancing in DSs – Case study

V-System (Stanford)

V-System (Stanford)

- state-change driven information policy
  - significant change in CPU/memory utilization is broadcast to all other nodes
- *M* least loaded nodes are receivers, others are senders
- sender-initiated with new job selection policy
- *Location policy:*
  - probe random receiver
  - if still receiver (below the threshold), transfer the job
  - otherwise try another

Scheduling/Load-balancing in DSs – Case study

Sprite (Berkeley) I.

Sprite (Berkeley)

- *Centralized information policy:* coordinator keeps info
  - state-change driven information policy
  - Receiver: workstation with no keyboard/mouse activity for the defined time period (30 seconds) and below the limit (active processes $<$ number of processors)
- *Selection policy:* manually done by user $\Rightarrow$ workstation becomes sender
- *Location policy:* sender queries coordinator
- the workstation with the foreign process becomes sender if user becomes active

Scheduling/Load-balancing in DSs – Case study

Sprite (Berkeley) II.

Sprite (Berkeley) cont'd.

- *Sprite process migration:*
  - facilitated by the Sprite file system
  - state transfer:
    - swap everything out
    - send page tables and file descriptors to the receiver
    - create/establish the process on the receiver and load the necessary pages
    - pass the control
  - the only problem: communication-dependencies
    - solution: redirect the communication from the workstation to the receiver

Scheduling/Load-balancing in DSs – Code and Process Migration

Code and Process Migration

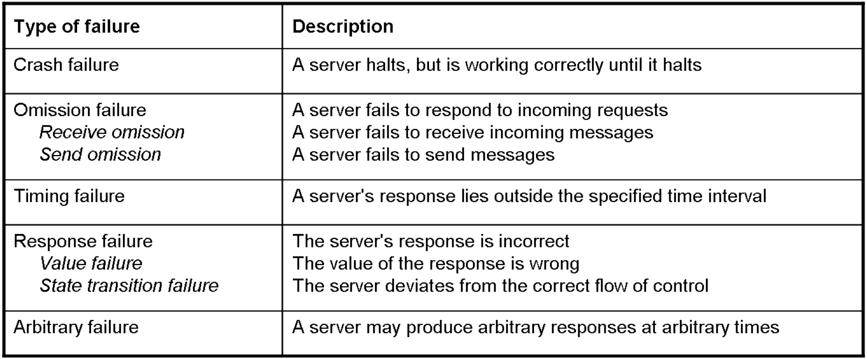
- key reasons: *performance* and *flexibility*
- flexibility:
  - dynamic configuration of distributed system
  - clients don't need preinstalled software (download on demand)
- process migration (*strong mobility*)
  - process = code + data + stack
  - examples: Condor, DQS
- code migration (*weak mobility*)
  - transferred program always starts from its initial state
- *migration in heterogeneous systems:*
  - only weak mobility is supported in common systems (recompile code, no run time information)
  - the virtual machines may be used: interprets (scripts) or intermediate code (Java)

Fault Tolerance in Distributed Systems I.

- single machine systems
  - failures are all or nothing
    - OS crash, disk failures, etc.
- distributed systems: multiple independent nodes
  - partial failures are also possible (some nodes fail)
  - probability of failure grows with number of independent components (nodes) in the system
- **fault tolerance:** system should provide services despite faults
  - *transient faults*
  - *intermittent faults*
  - *permanent faults*

Fault Tolerance in Distributed Systems I.

Failure Types



Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Fault Tolerance in Distributed Systems II.

- *handling faulty processes:* through redundancy
- organize several processes into a group
  - all processes perform the same computation
  - all messages are sent to all the members of the particular group
  - majority needs to agree on results of a computation
  - ideally, multiple independent implementations of the application are desirable (to prevent identical bugs)
- use process groups to organize such processes

Fault Tolerance in Distributed Systems III.

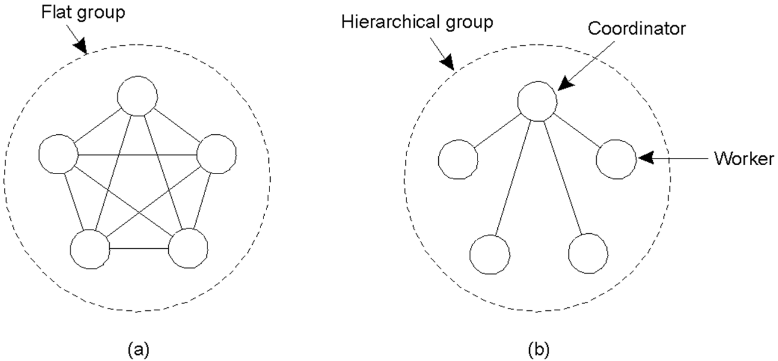


Figure: Flat Groups vs. Hierarchical Groups.

Fault Tolerance in DSs – Agreement in Faulty Systems

- How should processes agree on results of a computation?
- *K-fault tolerant:* system can survive k faults and yet function
  - assume processes fail silently
    - $\Rightarrow$ need $(k+1)$ redundancy to tolerant k faults
- *Byzantine failures:* processes run even if sick
  - produce erroneous, random or malicious replies
  - byzantine failures are most difficult to deal with

Fault Tolerance in DSs – Agreement in Faulty Systems

Byzantine Faults

Byzantine Generals Problem:

- four generals lead their divisions of an army
  - the divisions camp on the mountains on the four sides of an enemy-occupied valley
- the divisions can only communicate via messengers
  - messengers are totally reliable, but may need an arbitrary amount of time to cross the valley
    - they may even be captured and never arrive
- if the actions taken by each division is not consistent with that of the others, the army will be defeated
- we need a scheme for the generals to agree on a common plan of action (attack or retreat)
  - even if some of the generals are traitors who will do anything to prevent loyal generals from reaching the agreement
- the problem is nontrivial even if messengers are totally reliable
  - with unreliable messengers, the problem is very complex

Fault Tolerance in DSs – Agreement in Faulty Systems

Formal Definition I.

Formal definition of the agreement problem in DSs:

- let's have a set of distributed processes with initial states $\in 0, 1$
- *the goal:* all the processes have to agree on the same value
  - additional requirement: it must be possible to agree on both 0 or 1 states
- basic assumptions:
  - *system is asynchronous*
    - no bounds on processes' execution delays exist
    - no bounds on messages' delivery delay exist
    - there are no synchronized clocks
  - *no communication failures* – every process can communicate with its neighbors
  - *processes fail by crashing* – we do not consider byzantine failures

Fault Tolerance in DSs – Agreement in Faulty Systems

Formal Definition II.

Formal definition of the agreement problem in DSs: cont'd.

- *implications:*
  - ⇒ there is no deterministic algorithm which resolves the consensus problem in an asynchronous system with processes, which may fail
  - because it is impossible to distinguish the cases:
    - a process does not react, because it has failed
    - a process does not react, because it is slow
  - practically overcomed by establishing timeouts and by ignoring/killing too slow processes
    - timeouts used in so-called Failure Detectors (see later)

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast

Fault-tolerant Broadcast:

- *if there was a proper type of fault-tolerant broadcast, the agreement problem would be solvable*
- various types of broadcasts:
  - *reliable broadcast*
  - *FIFO broadcast*
  - *casual broadcast*
  - *atomic broadcast* – the broadcast, which would solve the agreement problem in asynchronous systems

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Reliable Broadcast

Reliable Broadcast:

- basic features:
    - *Validity* – if a correct process broadcasts *m*, then it eventually delivers *m*
    - *Agreement* – if a correct process delivers *m*, then all correct processes eventually deliver *m*
    - *(Uniform) Integrity* – *m* is delivered by a process at most once, and only if it was previously broadcasted
- possible to implement using `send`/`receive` primitives:
    - the process *p* sending the broadcast message marks the message by its identifier and sequence number
        - and sends it to all its neighbors
    - once a message is received:
        - if the message has not been previously received (based in sender's ID and sequence number), the message is delivered

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – FIFO Broadcast

FIFO Broadcast:

- the reliable broadcast cannot assure the messages' ordering
  - it is possible to receive a subsequent message (from the sender's view) before the previous one is received
- *FIFO broadcast:* the messages from a single sender have to be delivered in the same order as they were sent
- FIFO broadcast $=$ Reliable broadcast $+$ FIFO ordering
  - if a process p broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m
  - $broadcast_p(m) \rightarrow broadcast_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$
- a simple extension of the reliable broadcast

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Casual Broadcast

Casual Broadcast:

- the FIFO broadcast is still not sufficient: it is possible to receive a message from a third party, which is a reaction to a particular message before receiving that particular message
  - $\Rightarrow$ *Casual broadcast*
- Casual broadcast = Reliable broadcast + casual ordering
  - if the broadcast of a message *m happens before* the broadcast of a message m', then no correct process delivers m' unless it has previously delivered *m*
  - $broadcast_p(m) \rightarrow broadcast_q(m') \Rightarrow deliver_r(m) \rightarrow deliver_r(m')$
- can be implemented as an extension of the FIFO broadcast

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Atomic Broadcast

Atomic Broadcast:

- even the casual broadcast is still not sufficient: sometimes, it is necessary to guarantee the proper in-order delivery of all the replicas
  - two bank offices: one of them receives the information about adding an interest before adding a particular amount of money to the account, the second one receives these messages contrariwise
    - ⇒ inconsistency
  - ⇒ *Atomic broadcast*
- Atomic broadcast = Reliable broadcast + total ordering
  - if correct processes p and q both deliver messages m, m', then p delivers m before m' if and only if q delivers m before m'
  - $deliver_p(m) \rightarrow deliver_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$
- does not exist in asynchronous systems

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Timed Reliable Broadcast

Timed Reliable Broadcast:

- a way to practical solution
- introduces an upper limit (time), before which every message has to be delivered
- Timed Reliable broadcast $=$ Reliable broadcast $+$ timeliness
  - there is a known constant Δ such that if a message is broadcasted at real-time t, then no correct (any) process delivers m after real-time $t + \Delta$
- feasible in asynchronous systems
- A kind of “approximation” of atomic broadcast

Fault Tolerance in DSs – Agreement in Faulty Systems

Failure Detectors I.

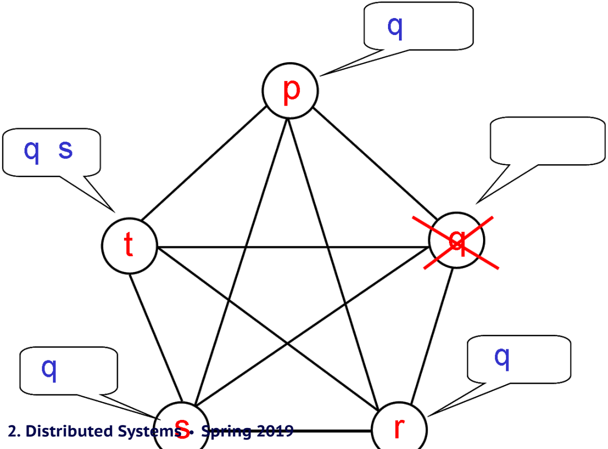
- impossibility of consensus caused by inability to detect slow process and a failed process
  - synchronous systems: let’s use timeouts to determine whether a process has crashed
  - ⇒ *Failure Detectors*

Failure Detectors (FDs):

- a distributed oracle that provides hints about the operational status of processes (which processes had failed)
  - FDs communicate via atomic/time reliable broadcast
- every process maintains its own FD
  - and asks just it to determine, whether a process had failed
- *however:*
  - hints may be incorrect
  - FD may give different hints to different processes

Fault Tolerance in DSs – Agreement in Faulty Systems

Failure Detectors II.



Fault Tolerance in DSs – Agreement in Faulty Systems

(Perfect) Failure Detector

Perfect Failure Detector:

- properties:
  - *Eventual Strong Completeness* – eventually every process that has crashed is permanently suspected by all non-crashed processes
  - *Eventual Strong Accuracy* – no correct process is ever suspected
- hard to implement
- is perfect failure detection necessary for consensus? **No.**
  - ⇒ weaker *Failure Detector*

weaker **Failure Detector:**

- properties:
  - *Strong Completeness* – there is a time after which every faulty process is suspected by every correct process
  - *Eventual Strong Accuracy* – there is a time after which no correct process is suspected

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Web Services

Grid Services

Issues Examples

- Scheduling/Load-balancing in Distributed Systems
- Fault Tolerance in Distributed Systems

Conclusion

Distributed Systems – Further Information

- **FI courses:**
  - PA150: Advanced Operating Sytems Concepts (doc. Staudek)
  - PA053: Distributed Systems and Middleware (doc. Tůma)
  - IA039: Supercomputer Architecture and Intensive Computations (prof. Matyska)
  - PA177: High Performance Computing (LSU, prof. Sterling)
  - IV100: Parallel and distributed computations (doc. Královič)
  - IB109: Design and Implementation of Parallel Systems (dr. Barnat)
  - etc.
- **(Used) Literature:**
  - W. Jia and W. Zhou. Distributed Network Systems: From concepts to implementations. Springer, 2005.
  - A. S. Tanenbaum and M. V. Steen. Distributed Systems: Principles and paradigms. Pearson Prencite Hall, 2007.
  - G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems: Concepts and design. Addison-Wesley publishers, 2001.
  - Z. Tari and O. Bukhres. Fundamentals of Distributed Object Systems: The CORBA perspective. John Wiley & Sons, 2001.