Chapter 01: Introduction
<table>
<thead>
<tr>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: Introduction</td>
</tr>
<tr>
<td>02: Architectures</td>
</tr>
<tr>
<td>03: Processes</td>
</tr>
<tr>
<td>04: Communication</td>
</tr>
<tr>
<td>05: Naming</td>
</tr>
<tr>
<td>06: Synchronization</td>
</tr>
<tr>
<td>07: Consistency &amp; Replication</td>
</tr>
<tr>
<td>08: Fault Tolerance</td>
</tr>
<tr>
<td>09: Security</td>
</tr>
<tr>
<td>10: Distributed Object-Based Systems</td>
</tr>
<tr>
<td>11: Distributed File Systems</td>
</tr>
<tr>
<td>12: Distributed Web-Based Systems</td>
</tr>
<tr>
<td>13: Distributed Coordination-Based Systems</td>
</tr>
</tbody>
</table>
Distributed System: Definition

A distributed system is a piece of software that ensures that:

\[ a \text{ collection of independent computers appears to its users as a single coherent system } \]

Two aspects: (1) independent computers and (2) single system \( \Rightarrow \) middleware.
Goals of Distributed Systems

- Making resources available
- Distribution transparency
- Openness
- Scalability
# Distribution Transparency

<table>
<thead>
<tr>
<th>Transp.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and invocation mechanisms</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where an object is located</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that an object may be moved to another location while in use</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that an object may move to another location</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that an object is replicated</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that an object may be shared by several independent users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide failure and possible recovery of an object</td>
</tr>
</tbody>
</table>

**Note**

Distribution transparency is a nice a goal, but achieving it is a different story.
# Distribution Transparency

<table>
<thead>
<tr>
<th>Transp.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and invocation mechanisms</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where an object is located</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that an object may be moved to another location while in use</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that an object may move to another location</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that an object is replicated</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that an object may be shared by several independent users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide failure and possible recovery of an object</td>
</tr>
</tbody>
</table>

**Note**

Distribution transparency is a nice goal, but achieving it is a different story.
Degree of Transparency

Observation

Aiming at full distribution transparency may be too much:

- Users may be located in different continents
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches exactly up-to-date with the master
  - Immediately flushing write operations to disk for fault tolerance
Degree of Transparency

Observation

Aiming at full distribution transparency may be too much:

- Users may be located in different continents
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches exactly up-to-date with the master
  - Immediately flushing write operations to disk for fault tolerance
Aiming at full distribution transparency may be too much:

- Users may be located in different continents
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches exactly up-to-date with the master
  - Immediately flushing write operations to disk for fault tolerance
Degree of Transparency

Observation

Aiming at full distribution transparency may be too much:

- Users may be located in different continents
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches exactly up-to-date with the master
  - Immediately flushing write operations to disk for fault tolerance
Open distributed system

Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined interfaces
- Systems should support portability of applications
- Systems should easily interoperate

Achieving openness

At least make the distributed system independent from heterogeneity of the underlying environment:

- Hardware
- Platforms
- Languages
Openness of Distributed Systems

Open distributed system

Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined interfaces
- Systems should support portability of applications
- Systems should easily interoperate

Achieving openness

At least make the distributed system independent from heterogeneity of the underlying environment:

- Hardware
- Platforms
- Languages
Policies versus Mechanisms

**Implementing openness**

Requires support for different policies:

- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

**Implementing openness**

Ideally, a distributed system provides only mechanisms:

- Allow (dynamic) setting of caching policies
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms
Policies versus Mechanisms

**Implementing openness**
Requires support for different policies:
- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

**Implementing openness**
Ideally, a distributed system provides only mechanisms:
- Allow (dynamic) setting of caching policies
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms
Scale in Distributed Systems

**Observation**

Many developers of modern distributed system easily use the adjective “scalable” without making clear **why** their system actually scales.

**Scalability**

At least three components:

- Number of users and/or processes *(size scalability)*
- Maximum distance between nodes *(geographical scalability)*
- Number of administrative domains *(administrative scalability)*

**Observation**

Most systems account only, to a certain extent, for size scalability. The *(non)solution*: powerful servers. Today, the challenge lies in geographical and administrative scalability.
Scale in Distributed Systems

Observation
Many developers of modern distributed system easily use the adjective “scalable” without making clear **why** their system actually scales.

Scalability
At least three components:

- Number of users and/or processes **(size scalability)**
- Maximum distance between nodes **(geographical scalability)**
- Number of administrative domains **(administrative scalability)**

Observation
Most systems account only, to a certain extent, for size scalability. The (non)solution: powerful servers. Today, the challenge lies in geographical and administrative scalability.
Observation
Many developers of modern distributed system easily use the adjective “scalable” without making clear why their system actually scales.

Scalability
At least three components:
- Number of users and/or processes (size scalability)
- Maximum distance between nodes (geographical scalability)
- Number of administrative domains (administrative scalability)

Observation
Most systems account only, to a certain extent, for size scalability. The (non)solution: powerful servers. Today, the challenge lies in geographical and administrative scalability.
Hide communication latencies

Avoid waiting for responses; do something else:

- Make use of asynchronous communication
- Have separate handler for incoming response
- **Problem:** not every application fits this model
Techniques for Scaling

**Distribution**

Partition data and computations across multiple machines:
- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)
Techniques for Scaling

**Replication/caching**

Make copies of data available at different machines:

- Replicated file servers and databases
- Mirrored Web sites
- Web caches (in browsers and proxies)
- File caching (at server and client)
Observation

Applying scaling techniques is easy, except for one thing:

- Having multiple copies (cached or replicated), leads to inconsistencies: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires global synchronization on each modification.
- Global synchronization precludes large-scale solutions.

Observation

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but tolerating inconsistencies is application dependent.
Scaling – The Problem

Observation
Applying scaling techniques is easy, except for one thing:

- Having multiple copies (cached or replicated), leads to **inconsistencies**: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires **global synchronization** on each modification.
- Global synchronization precludes large-scale solutions.

Observation
If we can tolerate inconsistencies, we may reduce the need for global synchronization, but **tolerating inconsistencies is application dependent.**
Scaling – The Problem

Observation
Applying scaling techniques is easy, except for one thing:

- Having multiple copies (cached or replicated), leads to 
inconsistencies: modifying one copy makes that copy different 
from the rest.
- Always keeping copies consistent and in a general way requires 
global synchronization on each modification.
- Global synchronization precludes large-scale solutions.

Observation
If we can tolerate inconsistencies, we may reduce the need for global 
synchronization, but tolerating inconsistencies is application 
dependent.
Scaling – The Problem

**Observation**

Applying scaling techniques is easy, except for one thing:
- Having multiple copies (cached or replicated), leads to **inconsistencies**: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires **global synchronization** on each modification.
- Global synchronization precludes large-scale solutions.

**Observation**

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but **tolerating inconsistencies is application dependent**.
Scaling – The Problem

**Observation**

Applying scaling techniques is easy, except for one thing:

- Having multiple copies (cached or replicated), leads to **inconsistencies**: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires **global synchronization** on each modification.
- Global synchronization precludes large-scale solutions.

**Observation**

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but **tolerating inconsistencies is application dependent**.
Developing Distributed Systems: Pitfalls

Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Developing Distributed Systems: Pitfalls

**Observation**

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. There are many false assumptions:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator
Types of Distributed Systems

- Distributed Computing Systems
- Distributed Information Systems
- Distributed Pervasive Systems
Distributed Computing Systems

Observation
Many distributed systems are configured for High-Performance Computing

Cluster Computing
Essentially a group of high-end systems connected through a LAN:
- Homogeneous: same OS, near-identical hardware
- Single managing node
Distributed Computing Systems

1.3 Types of Distributed Systems

- **Distributed Computing Systems**
  - Local OS
  - Standard network
  - Remote access network
  - Master node
    - Management application
    - Parallel libs
    - Local OS
  - Compute node
    - Component of parallel application
    - Local OS
  - Compute node
    - Component of parallel application
    - Local OS
  - Compute node
    - Component of parallel application
    - Local OS
- **Networks**
  - Standard network
  - High-speed network

Grid Computing

The next step: lots of nodes from everywhere:

- Heterogeneous
- Dispersed across several organizations
- Can easily span a wide-area network

Note

To allow for collaborations, grids generally use virtual organizations. In essence, this is a grouping of users (or better: their IDs) that will allow for authorization on resource allocation.
Distributed Computing Systems: Clouds

- **Application**
  - Web services, multimedia, business apps

- **Platforms**
  - Software framework (Java/Python/.Net)
  - Storage (DB, File)

- **Infrastructure**
  - Computation (VM), storage (block)

- **Hardware**
  - CPU, memory, disk, bandwidth

- **Datacenters**
  - MS Azure
  - Amazon S3
  - Amazon EC2

- **Web Services**
  - Google Apps
  - YouTube
  - Flickr
Distributed Computing Systems: Clouds

**Cloud computing**

Make a distinction between four layers:

- **Hardware**: Processors, routers, power and cooling systems. Customers normally never get to see these.
- **Infrastructure**: Deploys virtualization techniques. Evolves around allocating and managing virtual storage devices and virtual servers.
- **Platform**: Provides higher-level abstractions for storage and such. Example: Amazon S3 storage system offers an API for (locally created) files to be organized and stored in so-buckets.
- **Application**: Actual applications, such as office suites (text processors, spreadsheet applications, presentation applications). Comparable to the suite of apps shipped with OSes.
Distributed Information Systems

Observation

The vast amount of distributed systems in use today are forms of traditional information systems, that now integrate legacy systems. Example: Transaction processing systems.

BEGIN_TRANSACTION(server, transaction)
READ(transaction, file-1, data)
WRITE(transaction, file-2, data)
n newData := MODIFIED(data)
IF WRONG(newData) THEN
   ABORT_TRANSACTION(transaction)
ELSE
    WRITE(transaction, file-2, newData)
END_TRANSACTION(transaction)
END IF
Distributed Information Systems

**Observation**

The vast amount of distributed systems in use today are forms of traditional information systems, that now integrate legacy systems. **Example**: Transaction processing systems.

```plaintext
BEGIN_TRANSACTION(server, transaction)
READ(transaction, file-1, data)
WRITE(transaction, file-2, data)
newData := MODIFIED(data)
IF WRONG(newData) THEN
    ABORT_TRANSACTION(transaction)
ELSE
    WRITE(transaction, file-2, newData)
END_TRANSACTION(transaction)
END IF
```

**Note**

Transactions form an atomic operation.
Distributed Information Systems: Transactions

**Model**

A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID)

**Atomicity:** All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

**Consistency:** A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction’s execution.

**Isolation:** Concurrent transactions do not interfere with each other. It appears to each transaction $T$ that other transactions occur either before $T$, or after $T$, but never both.

**Durability:** After the execution of a transaction, its effects are made permanent: changes to the state survive failures.
Distributed Information Systems: Transactions

Model

A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):

**Atomicity:** All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

**Consistency:** A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction’s execution.

**Isolation:** Concurrent transactions do not interfere with each other. It appears to each transaction $T$ that other transactions occur either before $T$, or after $T$, but never both.

**Durability:** After the execution of a transaction, its effects are made permanent: changes to the state survive failures.
Distributed Information Systems: Transactions

**Model**

A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):

**Atomicity**: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

**Consistency**: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction’s execution.

**Isolation**: Concurrent transactions do not interfere with each other. It appears to each transaction $T$ that other transactions occur either before $T$, or after $T$, but never both.

**Durability**: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.
Distributed Information Systems: Transactions

Model

A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID)

Atomicity: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

Consistency: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction’s execution.

Isolation: Concurrent transactions do not interfere with each other. It appears to each transaction \( T \) that other transactions occur either before \( T \), or after \( T \), but never both.

Durability: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.
A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):

**Atomicity:** All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

**Consistency:** A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction’s execution.

**Isolation:** Concurrent transactions do not interfere with each other. It appears to each transaction $T$ that other transactions occur either before $T$, or after $T$, but never both.

**Durability:** After the execution of a transaction, its effects are made permanent: changes to the state survive failures.
In many cases, the data involved in a transaction is distributed across several servers. A **TP Monitor** is responsible for coordinating the execution of a transaction.
A TP monitor doesn’t separate apps from their databases. Also needed are facilities for direct communication between apps.

- Remote Procedure Call (RPC)
- Message-Oriented Middleware (MOM)
**Observation**

Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system.

**Some requirements**

- **Contextual change**: The system is part of an environment in which changes should be immediately accounted for.
- **Ad hoc composition**: Each node may be used in a very different ways by different users. Requires ease-of-configuration.
- **Sharing is the default**: Nodes come and go, providing sharable services and information. Calls again for simplicity.

**Note**

Pervasiveness and distribution transparency: a good match?
Distributed Pervasive Systems

Observation
Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system.

Some requirements
- **Contextual change**: The system is part of an environment in which changes should be immediately accounted for.
- **Ad hoc composition**: Each node may be used in a very different ways by different users. Requires ease-of-configuration.
- **Sharing is the default**: Nodes come and go, providing sharable services and information. Calls again for simplicity.

Note
Pervasiveness and distribution transparency: a good match?
Pervasive Systems: Examples

**Home systems**

Should be completely self-organizing:
- There should be no system administrator
- Simplest solution: a centralized *home box*?

**Monitoring a person**

Devices are physically close to a person:
- Where and how should monitored data be stored?
- How can we prevent loss of crucial data?
- What is needed to generate and propagate alerts?
- How can security be enforced?
- How can environment provide online feedback?
**Pervasive Systems: Examples**

**Home systems**

Should be completely self-organizing:
- There should be no system administrator
- Simplest solution: a centralized home box?

**Monitoring a person**

Devices are physically close to a person:
- Where and how should monitored data be stored?
- How can we prevent loss of crucial data?
- What is needed to generate and propagate alerts?
- How can security be enforced?
- How can environment provide online feedback?
Sensor networks

**Characteristics**

The nodes to which sensors are attached are:

- Many (10s-1000s)
- Simple (small memory/compute/communication capacity)
- Often battery-powered (or even battery-less)
Sensor networks as distributed systems

(a) Sensor data is sent directly to the operator.

(b) Each sensor can process and store data.
Speculative examples....

**Crowd management**

- **Situation**: a large event without fixed routes (trade fair, festival)
- **Goal**: guide people according to their social position:
  - Direct people having similar interests to the same locations
  - Direct the same social group to the same exit in the event of an emergency
- **Goal**: keep groups together (such as a family)
Application scenarios: social games

**Stimulate mingling**

- **Situation**: Conference with people from different groups.
- **Goal**: Stimulate people from different groups to interact.
- **Approach**: Keep track of group interactions:
  - When Alice from VU talks to Sajith from Amrita: bonus points for either one, as well as their respective groups.
  - Group points are distributed to each member.
  - Mingling achievements are displayed on electronic badges (feedback and social intervention).
Application scenarios: social games