On Building Dependable Distributed Applications

Gianpaolo Cugola
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

cugola@elet.polimi.it
http://home.dei.polimi.it/cugola
Outline

• Dependability in distributed applications
• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang

On building dependable distributed systems
Motivations

• Building standard (i.e., single user, single host) applications is not easy
  – Building “dependable” standard applications is hard
• Distribution raises this complexity by orders of magnitude
• Building distributed applications is really hard
  – Building dependable ones is…
  – … sorry, I finished possible adjectives
The various faces of dependability

• What is dependability
  – Availability
    • We want the system to be ready for use as soon as we need it
  – Reliability
    • The system should be able to run continuously for long time
  – Safety
    • It should cause nothing bad to happen
  – Maintainability
    • It should be easy to fix

• Availability vs. Reliability
  – If a system goes down for one millisecond every hour, it has an availability of 99.999%, but is still highly unreliable
Dependability @ Google

- “Google is technologically a large supercomputer. It's a distributed supercomputer among many data centers doing all sorts of interesting things over fiber optic network that eventually are services available to end-users.” - Eric Schmidt, Google CEO, 2007

- “Google runs on hundreds of thousands of servers—by one estimate today they are more than 2’000’000—racked up in thousands of clusters in dozens of data centers around the world.”

- “A typical search will require actions from between 700 to 1’000 machines today.” - Marissa Mayer, vice president of Google’s search products and user experience, 2008

- “Our view is it's better to have twice as much hardware that's not as reliable than half as much that's more reliable...You have to provide reliability on a software level. If you're running 10,000 machines, something is going to die every day.” - Jeff Dean, Google fellow, 2008

  The web site monitoring service Pingdom tracked Google’s worldwide network of search sites for a one-year period ending in October 2007, and found that all 32 of Google’s worldwide search portals (including google.co.uk, google.in, etc.) maintained uptime of at least “four nines” – 99.99 percent. The main site at google.com was down for 31 minutes in the 12-month monitoring period. The best performer was Google Brazil (google.com.br), with 3 minutes of downtime.
Pervasiveness: An Internet of Things

- “The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it... only when things disappear in this way are we freed to use them without thinking and so to focus beyond them on new goals.” – Mark Weiser, 1991

- “... by embedding short-range mobile transceivers into a wide array of gadgets and everyday items..., a new dimension has been added to the world of information and communication technologies: from anytime, any place connectivity for anyone, we will now have connectivity for anything.” – The Internet of Things, ITU Internet Reports 2005.

- “In the future, everything of value will be on the network in one form of another,” - John Fowler, Software CTO of Sun Microsystems.
Wireless sensor networks

On building dependable distributed systems
Internet performance

Source: Internet Traffic Report

On building dependable distributed systems
On building dependable distributed systems
Positive issues

• Standard, centralized systems typically fails completely
  – If the HD fails there is not much your application can do… but crash

• Distributed systems fail only partially
  – Each component fails independently, leaving the others still running
  – Most failures happen in communication channels

• This is good and bad
  – You can be fault tolerant…
  – …you must be fault tolerant
    • Google CEO cannot say: “sorry we are down due to a motherboard failing”, or “sorry we loose your email, one of our HDs failed”
Dependability and centralization

• Centralization is bad
  – Single point of failure

• Typical points of centralization:
  – Services
    • E.g., a single server for all users
  – Data
    • E.g., a single on-line telephone book
  – Algorithms
    • E.g., doing routing based on complete information

• Decentralized algorithms:
  – No machine has complete information about the system state
  – Machines make decisions based only on local information
  – Failure of one machine does not ruin the algorithm
  – There is no implicit assumption that a global clock exists

• Several techniques available
  – Asynchronous communication, caching and replication, epidemic dissemination, hierarchical structuring, …

• Decentralization also helps with scalability
  – Centralization ↔ single point of failure ↔ bottleneck

On building dependable distributed systems
Outline

• Dependability in distributed applications

• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang
How to build dependable distributed applications

- Building dependable distributed applications is hard, not impossible
- A lot of aspects to consider…
- … the key one is the system architecture you choose
  - The more decoupled the better
  - “Flexibility” and “re-configurability” are the keywords
- Second one is the choice of the right middleware technology that fits such architecture…
- …and helps in reducing implementation complexity
Middleware provides horizontal services to help building distributed applications. It also masks platform differences.
Middleware: A functional view

- Middleware provides “business-unaware” services through a standard API
- Usually it provides
  - Communication and coordination services
    - Synchronous and asynchronous
    - Point-to-point or multicast
    - Masking differences in the network OS
  - Special application services
    - Distributed transaction management, groupware and workflow services, messaging services, notification services, ...
  - Management services
    - Naming, security, failure handling, ...

On building dependable distributed systems
The run-time (system) architecture of a distributed system

- Identifies the classes of components that build the system, the various types of connectors, and the data types exchanged at run-time
- Modern distributed systems often adopt one among a small set of well known architectural styles
- We will see some of them
  - Client-server
  - Service Oriented
  - Object-oriented
  - Queue-based
  - Event-based
  - Mobile code
Outline

- Dependability in distributed applications
- The role of the architecture and middleware
  - The client/server case
    - MPI: A high level messaging API
  - The Service Oriented Architecture
    - Web services
  - The Object-Oriented Architecture
    - Java RMI
  - The queue-based and event based architectures
    - JMS and CEP
  - Mobile code
    - The role of the programming paradigm: Mobile code in Erlang
The client-server architecture

- The most common architectural style today
- Components have different roles
  - *Servers* provide a set of services through a well defined API
    - They are passive (just wait for client invocations)
  - Users access those services through *clients*
  - Communication is message based (or RPC)
What’s wrong with c/s

- Strong coupling between client and server
  - In space
    - Need to know the host that offers the service…
    - … and the exact address of such host
    - Such binding usually happens at the beginning of the interaction and does not change
  - In time
    - Interaction between client and server is synchronous

- Some consequences
  - Hard to replicate servers when required
  - Hard to relocate servers
  - Even client mobility is hard to support (requests & replies bound in space & time)
Possible workarounds

- Name and directory services help in reducing coupling in space (more on this later)
- Here we will (briefly) focus on using a middleware (MPI) to raise the level of abstraction of socket programming
  - Especially useful for supporting coordination among replicated servers and distribution of service execution
Datagram sockets: Fundamentals

- Sockets provide a low-level API for inter-process communication
  - The programming abstraction to access the services offered by TCP & UDP protocols
- Using datagram sockets client and server use the same approach to send and receive datagrams
  - Both create a socket bound to a port and use it to send and receive datagrams
- There is no connection and the same socket can be used to send (receive) datagrams to (from) multiple hosts
Datagram sockets in C

optional

\[
\begin{align*}
&\text{socket()} \\
&\quad\text{bind()} \\
&\quad\text{rcvfrom()} \\
&\quad\text{sendto()} \\
&\quad\text{close()}
\end{align*}
\]

\[
\begin{align*}
&\text{socket()} \\
&\quad\text{bind()} \\
&\quad\text{sendto()} \\
&\quad\text{recvfrom()} \\
&\quad\text{close()}
\end{align*}
\]
Multicast sockets

• IP multicast is a network protocol to efficiently deliver UDP datagrams to multiple recipients
  – The Internet Protocol reserve a class D address space, from 224.0.0.0 to 239.255.255.255, to multicast groups

• The socket API for multicast communication is similar to that for datagram communication
  – Component interested in receiving multicast datagrams addressed to a specific group must join the group (using the setsockopt call)
    • Groups are open: It is not necessary to be a member of a group in order to send datagrams to the group
  – As usual it is also necessary to specify a port
    • It is used by the OS to decide which process on the local machine to route packets to
Limitation of sockets
- Low level
- Protocol independent (and so awkward to use)

In high performance networks (e.g., clusters of computers) we need higher level primitives for asynchronous, transient communication...

...providing different services besides pure read and write

MPI was the (platform independent) answer
MPI: The model and main API

- Communication takes place within a known group of processes
- Each process within a group is assigned a local id
  - The pair (groupID, processID) represents a source or destination address
  - Messages can also be sent in broadcast to the entire group
- No support for fault tolerance (crashes are supposed to be fatal)
- Main MPI primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_bsend</td>
<td>Append outgoing message to a local send buffer</td>
</tr>
<tr>
<td>MPI_send</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_issend</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there is none</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>
Outline

• Dependability in distributed applications
• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang
The Service Oriented Architecture

• Built around the concepts of services, service providers, service consumers, service brokers
  – Services represent loosely coupled units of functionality...
  – ...exported by service providers
  – Brokers hold the description of available services to be searched by interested consumers...
  – ...which bind and invoke the services they need
  – Brokers helps in decoupling providers from consumers (servers from clients)

• Several incarnations
  – OSGI (Open Grid Services Infrastructure, JXTA, Jini, Web Services

On building dependable distributed systems
Web Services: A middleware for SOA

- *Web Service*: “a software system designed to support interoperable machine-to-machine interaction over a network” [W3C]

- Its interface is described *WSDL* (Web Service Description Language)
  - It includes the set of *operations* exported by the web service

- Web service operations are invoked through *SOAP*, a protocol, based on XML, which defines the way messages (operation calls) are actually exchanged
  - Usually based on HTTP but other transport protocols can be used

- *UDDI* (Universal Description Discovery & Integration) describes the rules that allows web services to be exported and searched through a *registry*
**Web services: Additional mechanisms**

- **WS-Policy**
  - A grammar for expressing capabilities, requirements, and general characteristics of Web services
  - Helps in searching for the “right” service

- **WS-ReliableMessaging**
  - Allow two systems to send messages between each other reliably, masking component/system/network failures

- **WS-Transactions**
  - Part of the WS-Coordination framework
  - Defines two coordination types:
    - Atomic Transactions: for individual operations
    - Business Activity: for long running transactions

- **WS-Eventing & WS-Notification**
  - Publish/subscribe for web services
  - Allows components to notify interested components when something relevant happens

On building dependable distributed systems
Outline

• Dependability in distributed applications
• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang
The object-oriented architecture

- The distributed components encapsulate a data structure providing an API (an **interface**) to access and modify it
  - Each component is responsible for ensuring the integrity of the data structure it encapsulates
  - The internal organization of such data structure (the **implementation**) is hidden to the other components (who may access it only through the API mentioned above)

- Components interact through RPC
- Its a “peer to peer” model
  - But its often used to implement client-server applications

- Pros
  - Information hiding hides complexity in accessing/managing the shared data
  - Encapsulation plus information hiding reduce the management complexity
    - E.g., the objects that build the server may be moved at run-time to share the load
  - Objects are easy to reuse among different applications
  - Legacy components can be wrapped within objects and easily integrated in new applications
RMI in general: A middleware for OO distributed systems

• Allows method invocations to happen among distributed objects
  – The aim is to obtain the advantages of OOP also in the distributed setting

• Shares many of the core concepts and mechanisms with RPC
  – Sometimes built on top of an RPC layer

• Various incarnations: Java RMI, CORBA, …
Interface definition language

- The IDL separates the interface from the implementation of a distributed object
  - To handle platform/language heterogeneity
- Such separation is one of the basic OO principles
- The IDLs for distributed objects usually support inheritance and exception handling
Java RMI

• The simplest among modern mainstream distributed object systems
  – Focuses only on remote method invocation
  – More advanced services provided by other components of the Java family (e.g., Jini, JNDI, …)

• Part of Java since version 1.1

• Innovative and/or relevant aspects:
  – Semantics of parameter passing
    • By copy & by reference
  – Class downloading

• Uses Java as an IDL (Remote interfaces)
The registry: A naive naming service

- The default lookup service, rmiregistry, provides for dynamic service binding
  - Maintains the association between a symbolic name and an object bound on the server side
  - The service is not distributed
  - Due to security concerns it is not possible to bind an object on a registry executing on a different host
- RMI servers bind their remote objects (a proxy is stored in registry)
- RMI clients can:
  - Obtain a remote object reference (proxy), given the symbolic name
  - Ask for the list of available names
- The class java.rmi.Naming is used to interact with the registry
  - Main methods: lookup, list, bind, rebind, unbind
Java Naming and Directory Interface (JNDI) provides a standard interface for Java applications to access a hierarchical naming service.

- **Supports:**
  - Binding: Associating names with objects
  - List & Lookup: Find an object based on a name

- **Names are organized as a hierarchy of NamingContext**
  - Each contains zero or more bindings toward other NamingContext or toward application objects
JNDI: Example

- Example: namespace organization by geographic region, then by department
- Each shadowed box is implemented by a NamingContext object
- NamingContext objects are traversed to locate a particular name
- For example, the logical name California/Manufacturing/Orders can be used to locate the Order object
Problem: Consider an invocation \( m(T \ a) \)
- We can expect that the bytecode for \( T \) is co-located with the remote object
- But, due to polymorphism, the client may send a subtype of \( T \)
  - Always the case when \( T \) is an interface
  - Same problem regardless of whether \( a \) is a remote object or not
    - Finding the stub class vs. the actual class

RMI is designed as an open system: cannot assume all possible types are preloaded everywhere

Solution: applications can annotate types with codebase information
- I.e., where to find the bytecode for a given type name
  - Typically through a Web server

Implemented by redefining the class loader and the serialization mechanism
**Java RMI: Some considerations**

- RMI does not fully mask distribution
  - Semantics of parameter passing
  - Remote exceptions
  - Remote interfaces

- This is a result of a precise choice, to distinguish local from remote interaction

- RMI is less powerful than other distributed object systems:
  - Meant to provide a fundamental building block instead of a complete solution
  - Other Java components build upon it (e.g., Jini, J2EE, …)

- The ability to dynamically download code provides a strong form of flexibility
Outline

• Dependability in distributed applications
• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang
The queue-based architecture

- Components collaborate by putting messages in queues and reading messages from queues
  - Queues store messages until retrieved
  - Each component may have multiple queues
  - Each queue can be accessed by several components

- Communication is:
  - Purely message based
  - Asynchronous
  - Point-to-point (to/from queues)
  - Persistent

On building dependable distributed systems
**Client-Server with queues**

- Clients send requests to the server’s queue
- The server asynchronously fetches requests, processes them, and returns results in the clients’ queues
  - Thanks to persistency and asynchronicity, clients need not remain connected
  - Queue sharing simplifies load balancing
The event-based architecture

- Components collaborate by exchanging information about occurrent *events*. In particular:
  - Components *publish* notifications about the events they observe, or
  - they *subscribe* to the events they are interested to be notified about

- Communication is:
  - Purely message based
  - Asynchronous
  - Multicast
  - Implicit
  - Anonymous

On building dependable distributed systems
MOM in general: A middleware for queue and event-based systems

- RPC/RMI foster a synchronous model
  - Natural programming abstraction, but:
    - Supports only point-to-point interaction
    - Synchronous communication is expensive
    - Intrinsically tight coupling between caller and callee, leads to “rigid” architectures

- Asynchronous communication models:
  - Often centered around the notion of message/event
    - Or persistent communication
  - Often supporting multi-point interaction
  - Brings more decoupling among components
**MOM: Types of communication**

- **Synchronous vs. asynchronous**
  - Synchronous: the sender is blocked until the recipient has stored (or received, or processed) the message
  - Asynchronous: the sender continues immediately after sending the message

- **Transient vs. persistent**
  - Transient: sender and receiver must both be running for the message to be delivered
  - Persistent: the message is stored in the communication system until it can be delivered

- **Several alternatives (and combinations) are provided in practice**
Transient communication

Pure Asynchronous
- A sends message and continues
- Message can be sent only if B is running
- B receives message

Receipt-based synchronous
- Send request and wait until received
- Request is received
- Process request
- ACK

Delivery-based synchronous
- Send request and wait until accepted
- Request is received
- Accepted
- Running, but doing something else
- Process request

Response-based synchronous
- Send request and wait for reply
- Request is received
- Accepted
- Running, but doing something else
- Process request
Persistent communication

Asynchronous

A sends message and continues

A stopped running

B is not running

B starts and receives message

Synchronous

A sends message and waits until accepted

Message is stored at B's location for later delivery

A stopped running

B is not running

B starts and receives message

On building dependable distributed systems
Different MOM flavors

• Message oriented communication comes in two flavors: Message queuing and publish-subscribe systems

• Several common characteristics
  – Both are “message oriented”
  – Both offer a strong decoupling among components
  – Both are often based on a set of “servers” to route messages from sender to recipients and/or to support persistency

• But also a lot of differences
  – Point-to-point persistent vs multipoint transient
Queuing: Communication primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>

```
program A
begin
    ...
    attach_queues();
    ...
    loop
        put(msg);
        ...
        ...
    end
get(msg);
end
end
```

```
program B
begin
    ...
    attach_queues();
    ...
    loop
        get(msg);
        ...
        ...
    end
put(msg);
end
end
```

On building dependable distributed systems
Message queueing: Architectural Issues

- Queues are identified by symbolic names
  - Need for a lookup service, possibly distributed, to convert queue-level addresses in network addresses
  - Often pre-deployed static topology/naming
**Message queuing: Architectural Issues**

- Queues are manipulated by queue managers
  - Local and/or remote, acting as relays (a.k.a. applicative routers)
- Relays often organized in an overlay network
  - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
  - Improves fault tolerance
  - Provides applications with multi-point without IP-level multicast
Message queuing: Architectural Issues

- Message brokers provide application-level gateways supporting message conversion
  - Useful when integrating sub-systems
Publish-subscribe

- Application components can publish asynchronous event notifications, and/or declare their interest in event classes by issuing a subscription
  - extremely simple API: only two primitives (publish, subscribe)
  - event notifications are simply messages
- Subscriptions are collected by an event dispatcher component, responsible for routing events to all matching subscribers
  - Can be centralized or distributed
- Communication is:
  - Transiently asynchronous
  - Implicit
  - Multipoint
- High degree of decoupling among components
  - Easy to add and remove components
  - Appropriate for dynamic environments
The expressiveness of the subscription language allows one to distinguish between:

- **Subject-based** (or topic-based)
  - The set of subjects is determined a priori
  - Analogous to multicast
  - e.g., subscribe to all events about “Distributed Systems”

- **Content-based**
  - Subscriptions contain expressions (event filters) that allow clients to filter events based on their content
  - The set of filters is determined by client subscriptions
  - A single event may match multiple subscriptions
  - e.g., subscribe to all events about a “Distributed System” class with date greater than 16.11.2004 and held in classroom D04

The two can be combined

Tradeoffs:

- Complexity of the implementation vs. expressiveness
- However, expressiveness allows additional filtering!
The event dispatcher

- In event-based systems, a special component of the architecture, the **event dispatcher**, is in charge of collecting subscriptions and routing event notifications based on such subscriptions.
  - For scalability reasons, its implementation can be distributed.
Architecture of the Dispatcher

• Centralized
  – A single component is in charge of collecting subscriptions and forward messages to subscribers

• Distributed
  – A set of message brokers organized in an overlay network cooperate to collect subscriptions and route messages
  – The topology of the overlay network and the routing strategy adopted may vary
    • Acyclic vs. cyclic overlay
The Java Message Service (JMS) defines an API to allow for creation, sending, receiving and reading messages belonging to an enterprise system

- Strives for portability among different platforms
- Tries to integrate seamlessly with the Java language

First specification appeared in 1998

JMS core concepts:
- **JMS provider**: The entity that implements the JMS API for a messaging product
- **JMS client**: The applicative software that uses the services of a JMS provider through the JMS API
- **JMS domains**: Either point-to-point (queue based) or publish-subscribe (topic based)
JMS domains

• The point-to-point domain:
  – Each message is addressed to a specific queue
  – Client extract messages from the queue(s) established to hold their messages

• The publish-subscribe domain is centered on a specific content-hierarchy (topics)
  – The system takes care of distributing the messages belonging to a specific part of the content hierarchy to all clients interested in that content

• JMS provides a set of domain-independent interfaces allowing communication in both domains under a common API
  – Usually called “common interfaces”
### JMS Architecture

- **JMS Clients**: Java programs sending and/or receiving messages
- **Non-JMS Clients**: Clients using the native API of the messaging system
- **Messages**: Data containers defined by the application to exchange information
- **JMS Provider**: The messaging product implementing the JMS API together with (possible) administrative and quality-control functionality
- **Administered Objects**: Preconfigured JMS objects created by an administrator and used by the clients
To allow for portability, the details regarding a specific JMS Provider must be hidden.

This is done by defining JMS administered objects:
- Created and customized by administrators
- Later accessed via portable JMS interfaces by clients

Two types of administered objects:
- ConnectionFactory: Used by a client to establish a connection to the JMS provider
- Destination: Used by a client to specify the destination of messages it will send
JMS Lookup

- Administered objects are placed in a JNDI namespace under well-known identifiers
- To establish a connection to a JMS provider, clients retrieves a `ConnectionFactory` from the JNDI namespace
JMS Object Relationships

- A **Connection** represents a link between the client and a JMS Provider.
- A **Destination** encapsulates the identity of a message destination.
- A **Session** is a single-threaded context for sending and receiving messages.
- Message producers and consumers send and receive messages in a **Session**, respectively.
Developing JMS applications

• A JMS client will typically:
  – Use JNDI to find a ConnectionFactory object
  – Use JNDI to find one or more Destination object(s)
    • Predefined identifiers can also be used to instantiate new Destinations
  – Use the ConnectionFactory to establish a Connection with the JMS Provider
  – Use the Connection to create one or more Session(s)
  – Combine Session(s) and Destination(s) to create the needed MessageConsumer and MessageProducer
  – Tell the Connection to start delivery of messages
Example: The point-to-point domain

```java
import javax.jms.*; import javax.naming.*;

public class Sender {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        QueueConnectionFactory qcf = (QueueConnectionFactory) ictx.lookup("qcf");
        ictx.close();
        QueueConnection qc = qcf.createQueueConnection();
        QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueSender qsend = qs.createSender(queue);
        TextMessage msg = qs.createTextMessage();
        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            qsend.send(msg);
        }
        qc.close();
    }
}

import javax.jms.*; import javax.naming.*;

public class Receiver {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        QueueConnectionFactory qcf = (QueueConnectionFactory) ictx.lookup("qcf");
        ictx.close();
        QueueConnection qc = qcf.createQueueConnection();
        QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueReceiver qrec = qs.createReceiver(queue);
        TextMessage msg;
        qc.start();
        int i;
        for (i = 0; i < 10; i++) {
            msg = (TextMessage) qrec.receive();
            System.out.println("Msg received: " + msg.getText());
        }
        qc.close();
    }
}
```

On building dependable distributed systems
Example: The pub-sub domain

```java
import javax.jms.*; import javax.naming.*;

public class Publisher {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Topic topic = (Topic) ictx.lookup("topic");
        TopicConnectionFactory tcf = (TopicConnectionFactory)
            ictx.lookup("tcf");
        ictx.close();

        TopicConnection tc = tcf.createTopicConnection();
        TopicSession ts = tc.createTopicSession(true,
            Session.AUTO_ACKNOWLEDGE);
        TopicPublisher tpub = ts.createPublisher(topic);
        TextMessage msg = ts.createTextMessage();

        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            tpub.publish(msg);
        }

        ts.commit();
        tc.close();
    }
}

class MsgListener implements MessageListener {
    String id;
    public MsgListener() {id = "";}
    public MsgListener(String id) {this.id = id;}
    public void onMessage(Message msg) {
        TextMessage tmsg = (TextMessage) msg;
        try {
            System.out.println(id + " : " + tmsg.getText());
        } catch (JMSException je) {
            je.printStackTrace();
        }
    }
}

import javax.jms.*; import javax.naming.*;

public class Subscriber {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Topic topic = (Topic) ictx.lookup("topic");
        TopicConnectionFactory tcf = (TopicConnectionFactory)
            ictx.lookup("tcf");
        ictx.close();

        TopicConnection tc = tcf.createTopicConnection();
        TopicSession ts = tc.createTopicSession(true,
            Session.AUTO_ACKNOWLEDGE);
        TopicSubscriber tsub = ts.createSubscriber(topic);
        tsub.setMessageListener(new MsgListener());
        tc.start();
        System.in.read();
        tc.close();
    }
}
```

On building dependable distributed systems
Example: The common interfaces

import javax.jms.*; import javax.naming.*;

public class Producer {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        Topic topic = (Topic) ictx.lookup("topic");
        ConnectionFactory cf = (ConnectionFactory) ictx.lookup("cf");
        ictx.close();

        Connection cnx = cf.createConnection();
        Session sess = cnx.createSession(false,
                                           Session.AUTO_ACKNOWLEDGE);
        MessageProducer prod = sess.createProducer(null);
        TextMessage msg = sess.createTextMessage();

        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            prod.send(queue, msg);
            prod.send(topic, msg);
        }

        cnx.close();
    }
}

import javax.jms.*; import javax.naming.*;

public class Subscriber {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        Topic topic = (Topic) ictx.lookup("topic");
        ConnectionFactory cf = (ConnectionFactory) ictx.lookup("cf");
        ictx.close();

        Connection cnx = cf.createConnection();
        Session sess = cnx.createSession(false,
                                         Session.AUTO_ACKNOWLEDGE);
        MessageConsumer recv = sess.createConsumer(queue);
        MessageConsumer subs = sess.createConsumer(topic);

        recv.setMessageListener(new MsgListener("Queue"));
        subs.setMessageListener(new MsgListener("Topic"));

        cnx.start();
        System.in.read();
        cnx.close();
    }
}

On building dependable distributed systems
**More JMS concepts**

**JMS message model**

- The header part contains information used by both clients and providers to identify and route messages
  - All messages support the same set of header fields
  - Several fields are present, among them:
    - JMSMessageID contains a value that uniquely identify a message
    - JMSCorrelationID contains an identifier that links this message to another one (it can be used to link a query to a reply)
    - JMSTimestamp contains the time a message has been handed off to a JMS provider to be sent
    - JMSReplyTo contains a Destination supplied by a client where a reply to the message must be sent
    - JMSPriority contains the message’s priority, ten levels of priority can be defined, from 0 (lowest) to 9 (highest)
- Additional information can be possibly added using the property part
  - It can contain application-specific properties, standard properties defined by JMS (act as optional header fields), or provider-specific properties
  - Properties are `<string, value>` pairs, where `value` can be one of Java built-in types or String
  - When a message is received, its properties are in read-only mode
  - The method `getPropertyNames()` returns an `Enumeration` of all the property names
  - The main use of properties is to support customized message selection
- The body part specifies the actual message content
More JMS concepts
JMS message selectors

- Allow a JMS Client to customize the messages actually delivered, based on the application’s interests
- A message selector is a predicate (a String) on a message’s header and property values, whose syntax is derived from SQL’s conditional expression syntax
  - The selector is passed as a parameter when the Consumer is created
  - The message is delivered when the selector evaluates to true
  - Type checking must be taken into account
- Example:

```
myMsg.setIntProperty("Property1", 2);
myMsg.setStringProperty("Property2", "Hi");
myMsg.setFloatProperty("Property3", 3.4821);
myMsg.setStringProperty("Property4", "2");
```

<table>
<thead>
<tr>
<th>Message Selector</th>
<th>Matches?</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Property1 = 2 AND Property3 &lt; 4”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property3 NOT BETWEEN 2 AND 5”</td>
<td>No</td>
</tr>
<tr>
<td>“Property2 IN {’Ciao’, ’Hallo’, ’Hi’}”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property5 IS NULL”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property4 &gt; 1”</td>
<td>No</td>
</tr>
</tbody>
</table>
Going beyond pub-sub: Complex Event Processing

- CEP systems adds the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones
  - Recently a number of languages and systems have been proposed to support such architecture (both under the CEP and DSMS labels)
CEP: Open issues

• The rule language
  – Find a balance between expressiveness and processing complexity

• The processing engine
  – How to efficiently match incoming (primitive) events to build complex ones

• Distribution
  – How to distribute processing
    • Clustered vs. networked solutions
Our Proposal: TESLA

• “One size fits all” language does not exist
  – At least, we have to find it yet
• We started from the following requirements
  – Focus specifically on events
    • Not generic data processing
  – Define a clear and precise semantics
    • Issues of selection and consumption policies
    • General issue of formal semantics
  – Be expressive
    • Useful for applications
  – Keep it simple
    • Easy to read and write
Define \( CE(\text{Att}_1 : \text{Type}_1, \ldots, \text{Att}_n : \text{Type}_n) \)

From \( \text{Pattern} \)

Where \( \text{Att}_1 = f_1(\ldots), \ldots, \text{Att}_n = f_n(\ldots) \)

Consuming \( e_1, \ldots, e_m \)
Patterns in TESLA

- Selection of a single event
  - $A(x>10)$
  - Timer()
- Selection of sequences
  - $A(x>10)$ and each $B$ within 5 min from $A$
  - $A(x>10)$ and last $B$ within 5 min from $A$
  - $A(x>10)$ and first $B$ within 5 min from $A$
  - Generalization
    - n-first / n-last
  - Parameters can be added between events in a pattern
Negations and Aggregates

• Two kinds of negations:
  – Interval based:
    • A and last B
    within 3 min from A
    and not C between B and A
  – Time based:
    • A and not C within 3 min from A

• Similarly, two kinds of aggregates
  – Interval based
    • Use values appearing between two events
  – Time based
    • Use values appearing in a time interval
Iterations

• We believe that explicit operators for repetitions are difficult to use/understand
  – Especially when different selection/consumption policies are allowed
• We achieve the same goal using hierarchies of events
  – Complex events can be used to define (more) complex events
• Recurring schemes of repetitions
  – Macros!
Outline

• Dependability in distributed applications
• The role of the architecture and middleware
  – The client/server case
    • MPI: A high level messaging API
  – The Service Oriented Architecture
    • Web services
  – The Object-Oriented Architecture
    • Java RMI
  – The queue-based and event based architectures
    • JMS and CEP
  – Mobile code
    • The role of the programming paradigm: Mobile code in Erlang
Mobile code

- A style based on the ability of relocating the components of a distributed application at run-time
  - Only the code or both the code and the state
- Different models depending on the original and final location of resources, know-how (the code) and computational components (including the state of execution)
Mobile code paradigms

Client-Server

Remote evaluation

Code on demand

On building dependable distributed systems
Mobile code technologies

• *Strong mobility* is the ability of a system to allow migration of both the code and the execution state of an executing unit to a different computational environment
  – Very few systems (usually research based) provide it

• *Weak mobility* is the ability of a system to allow code movement across different computational environments
  – Provided by several mainstream systems including Java, .Net, the Web
Mobile code in practice

• Pros
  – The ability to move pieces of code (or entire components) at run-time provides a great flexibility to programmers
    • New versions of a component can be uploaded at run-time without stopping the application
    • Existing components can be enriched with new functionalities
    • New services can be easily added
    • Existing services can be adapted to the client needs

• Cons
  – Securing mobile code applications is a mess
  – What about the state of a component whose code is being changed?
The impact of the programming paradigm

Server in an imperative (OO) language

```java
void run() {
    while(true) {
        socket.receive(packet);
        data=parseData(packet);
        state.update(data);
    }
}
```

Server in a functional language (Erlang)

```erlang
server(State) ->
    receive
        Packet ->
        Data = parseData(Packet),
        S1 = update(State, Data),
        server(S1);
    end.
```

On building dependable distributed systems
Mobile code in practice

- What if you want to change the way incoming packets are parsed and data is stored internally?
  - In the OO case you have to find a way to move the old representation of the internal’s server state to the new representation…
  - … and you have to stop and restart the server “importing” the old state
  - In Erlang you will (automatically) use the new version of the code at the next tail recursion
Conclusions

• A distributed application may follow several architectural styles
  – Some are more coupled than others
  – The more coupling the less flexibility
  – Strong coupling hinders dependability

• Middleware helps implementing dependable/scalable distributed applications
  – Each middleware naturally suggests the adoption of a given architectural style