Introduction to Distributed Computing

Definitions

- **Distributed Application**
  
  "An application made up of distinct components that are physically located on different computer systems, connected by a network."

- **Distributed System**
  
  "A non-centralized network consisting of numerous possibly heterogeneous computers that can communicate with one another and that appear to users a single system."

What is a Distributed System?

- **System behavior spread across multiple nodes**
- **Needs**
  - Users – looks like a single system
  - Operator – monitor and control individual node or a collections of nodes
  - Developer – software infrastructure enables them to build components that deploy properly over entire system
  - Deployer – able to upgrade new software components without disrupting operational system
- **Characteristics**
  - Heterogeneous hardware, OSs, databases, networks, etc.
  - Scalable, available, fault tolerant, secure, transactional, etc.
  - Collection of related applications sharing common services and data
  - Evolve over time and must integrate with external systems

Distribution Relationships

- **Client-server**
  - Clients locate and directly communicate with server
    - Client aware of server and dependent on its existence
    - A client in one context may be a server in another
- **Peer-to-peer**
  - Services are peers and have capacity to act as both client and server
  - Peers less dependent on existence of specific peer
- **Broketed**
  - Client messages sent through a broker who determines routing, replication, etc.
Types of Communication

- **Custom data protocols**
  - Efficient, but tedious and hard to reuse services

- **Distributed objects**
  - Better abstraction, but tightly coupled in terms of connections and data types
  - Protocol standards exist – CORBA, RMI, .NET

- **Messaged-based communication**
  - Benefits of distributed objects without the tight coupling
  - More robust and flexible

Distribution System Architecture

- **N-tier systems provide facilities for presentation, business logic, and data components**
  - Vertical scaling running multiple components on a single node
  - Horizontally scaling running same component on multiple nodes
  - Systems historically in 1, 2, or 3-tier patterns

Distribution Issues

- **Managing distributed state**
  - Replicated components must synchronize state changes

- **Data marshaling**
  - Data transformed into format suitable for transmission of message
  - Can add significant overhead depending on format
  - Examples
    - CORBA CDR
    - Web Service's SOAP
    - Java serialization
    - .NET Remoting

1-tier Architecture

- **Entire application exists on single node**
  - Installed on individual machines

- **Advantages**
  - Simple to build and deploy

- **Disadvantages**
  - Facilitate very little reuse
  - Integration is extremely expensive, if not impossible
  - Single point of failure
  - Can only vertically scale, no horizontal scaling

Looks like a series of stovepipes
2-tier Architecture

- **Client-server applications**
  - Fat client vs. thin client
- **Advantages**
  - Modifications on server propagated to clients
  - Can distribute processing load and provide horizontal scaling
- **Disadvantages**
  - Client nodes require more computing power
  - Development and maintenance more complex
  - More developer and operator skills required

<table>
<thead>
<tr>
<th>Presentation</th>
<th>Logic</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Client</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Server</td>
<td>Server</td>
<td>Server</td>
</tr>
</tbody>
</table>

3-tier Architecture

- Each logical partition maps to a layer in the system
  - Modeling layers and software layers match 1-to-1
- Each layer implemented with appropriate technologies
  - Layers have their own internal architectures

**Presentation**

**Control**

**Data Mgmt**

3-tier Evaluation

- **Advantages**
  - Good separation of concerns
  - Control logic can be reused by client applications
  - Cache results in the middle layers for performance
  - Integrate legacy applications as services
- **Disadvantages**
  - Complexity increases cost of development and deployment
  - Performance decreases with levels of indirection and latency
  - Lack of knowledgeable developers, operators and managers

SOA Motivation

- **Integration drivers (Gartner)**
  - Information integration has always been important to IT, becoming important to business (board level)
  - Need higher quality, more reliable information across enterprise
- **Current state**
  - Applications hard-wired to resources
  - Hard to be agile and respond to changing business directions
  - Lots of multi-technology, point-to-point data integrations within organizations
    - Built for specific project using project’s money
    - Need enterprise focus
- **Solution**
  - Provide data services that expose information
    - Data, meta-information, aggregation, transformation, movement, access, synchronization
Managing Complexity and Integration

- How do others handle complexity and integration?
  - Services provided through ports with well-known communication standards
    - Provides easy integration, simple restructuring, component reuse, etc.
    - Black box implementations; communication does not imply implementation
    - Can broadcast to multiple endpoints
  - Operator can easily reconfigure, add/remove services, change service state, start/stop services, etc.

Services and Components

- What is a service?
  - Specifies course-grained, core business logic
  - Maintains no user state and returns result with each request (context free)
  - Can be reused across a diverse set of applications
  - Make no statement regarding implementation, deployment, etc.

- What is a component?
  - An individually deployable, executable code assembly
  - Commonly run inside a ‘container’, which provides lifecycle services and access to resources

Services and Components

- Service-Component relationship
  - Services may be realized by components
  - Components may use/depend on services

Client’s View of a Service

- Clients connect to services through their ports
- Ports specify operations which have input and output messages
- Messages specify typed data passed into and out of the service through
  - Type specifications decouple client from service provider

The above terms are used in Web Services, but the concepts generally apply to any SOA, particularly the separation of types
Service Oriented Architectures

- An architectural style achieving loose coupling among entities
  - Services are self-contained and not dependent on state of other services
    - Context-free – can be invoked without context of who is invoking
  - Communication achieved through normalized, asynchronous messages
- Loose coupling between service providers and consumers
  - Communication independent of implementation
- Not bound to Web services
  - However, Web Services are a common solution with a great deal of momentum
  - Web services infrastructure receiving a great deal of support from major technology companies

Principles of SOA

- Stateless interactions
  - Services are self-contained and do not store state between invocations
  - State should be managed in the business process (e.g. choreography) and provided to individual services
- Communicate through a normalized message broker (ESB)
  - Adapters may be required to convert legacy applications to participate
- Message-oriented, stateless communication vs. RPC
  - Focus is on the message, not the communication protocol
- Course-grained interfaces
  - Less learned from distributed objects – performance, scalability, etc.
- Location transparency
  - Clients have no knowledge of service provider’s location
  - Allows services on network to come and go without clients reconnecting

Basic Elements of SOAs

1. Communication infrastructure (transport)
   - The wire protocol for communication – e.g., SOAP/HTTP
2. Description of service specification
   - Specifies service, messages, responses, QoS guarantees, permissions, access control, etc.
3. Discovery and location
   - Allows clients to discover existing services
   - Criteria may include services properties (e.g., attributes, QoS)
4. Composition
   - Composes services to create higher-level business processes
   - Orchestration defines business and workflow processes
5. Additional infrastructure specifications
   - Security, transactions, event notification, etc.

1) Communications Infrastructure

- Transports message and data between client and service
  - Passes message and data and returns data result or fault
- Defines how clients denote service end points
  - Technology examples – remote object reference, URL, etc.
- Defines a normalized message format
  - Communications
  - Technology examples – IIOP, SOAP, etc.
2) Service Description

• Defines meta information describing service’s characteristics
  ➔ Information necessary to deploy and interact with a service

• Kinds of characteristics
  ➔ Functional characteristics define what messages the service sends and receives
  ➔ Policy information defines the service’s execution context
    ● Does service require security, transactions, etc.?
    ● May also define quality of service parameters, e.g., minimal encryption strength

• Technology examples
  ➔ CORBA interfaces and interface repository
  ➔ Web Service’s WSDL

3) Service Discovery

• Mechanism by which clients discover and bind to services
  ➔ Must uniquely identify service
  ➔ May also include meta-information about the service

• Technology examples
  ➔ Distributed object ‘naming services’ are most common example
  ➔ Web Services include UDDI, WS-Policy, DNS-Endpoint Discovery (the white pages for UDDI)

4) Service Composition

• Create a service from the composition and orchestration of lower-level services
  ➔ Orchestration defines centrally controlled process flow for services
  ➔ Choreography coordinates distributed services

• Technology examples – BPEL

5) Other Common Services

• Leasing
  ➔ Service usage granted for fixed period
    ● Allows service implementation to expire information it may be managing
    ● Allows SOA systems to self-heal when services and connections die

• Events
  ➔ Asynchronous communication eliminates need for polling

• Security, reliable messaging, transactions
  ➔ Provides common credentialing and state for these commonly used features