8. Development of Component Based Distributed Systems

Distributed Systems
Prof. Dr. Alexander Schill
http://www.rn.inf.tu-dresden.de
Outline

- Requirements engineering
- UML-based Specification
- Deployment based on Application Servers
- Distributed Debugging
Distributed Systems – Characteristics:
- Long lifespan
- Subject to constant change

Resulting design & lifecycle requirements:
- Readily changeable
- Maintainable and extendable

Tool support – required throughout lifecycle for:
- System modelling (requirements analysis and specification)
- Automation of routine tasks -> code generation, followed by implementation
- Deployment onto:
  - concrete component models
  - component platform technologies
  - executable component instances
- Monitoring and debugging at runtime
Requirements Analysis
Specification

UML
(Unified Modelling Language)

Generation / Implementation

Component Implementation

EJB / .NET / OSGi

Deployment

EJB Container, .NET, OSGi

Modified Requirements

Executed Component Instances

Installed Components

Instantiation

Component Specification
Requirements analysis

- Specification of structural and functional aspects:
  - User oriented concepts and terminology
    - Easy to understand
    - Abstraction from technical details
  - Readily changeable
  - Readily maintainable
  - Feasible to implement

Software Design & Lifecycle

Requirements Analysis

- UML (Unified Modelling Language)
- EJB / .NET / OSGi
- EJB Container, .NET, OSGi
Software Design & Lifecycle - Stages

- **Specification**
  - Whole system designed with a high degree of abstraction
  - Level of application architecture – independent of particular technologies and platforms
    - Components of coarse granularity
    - Interaction relationships
  - Step-by-step refinement of specification – decomposition to atomic components
    - e.g. order processing $\rightarrow$ shopping basket, product catalogue etc.
  - Modelling with UML
Application Modelling with UML

- **Unified Modelling Language – UML**
  - Based on object oriented concepts
    - Supports component based software design
    - Standard diagrammatic notation to describe:
      - Application structure
      - Interaction between components
      - Sequences of system states
      - Implementation details of components
  - Notation – Component Diagram

Diagram:

- **<<Component>> Order Processing**
- **<<Component>> Customer Administration**
- **<<Component>> Warehouse Management**
- **<<Component>> Dispatch Department**

Specification:

- UML (Unified Modelling Language)
- EJB / .NET / OSGi

Prof. Dr. Alexander Schill  Distributed Systems – Lecture 8: Development of Component Based Distributed Systems  Slide 7
Hierarchical Modelling

Order Processing component composed of further components. (See next slide)
Hierarchic Modelling – Component out of components

- **Ports** - map internal interfaces to elements in the next level up in the hierarchy.
Further Capabilities

• Diagrams
  • Sequence diagrams, Communication diagrams, state diagrams, distribution diagram

• Profiles
  • Tying to a concrete component model via extension of UML
  • E.g. UML Profile for EJB or OSGi

• Model Exchange
  • XML Metadata Interchange (XMI) – Standard internal representation of UML models
  • Exchange between different tool environments – for refinement of design via XML Metadata Interchange (XMI)

• Code Generation
  • Automatic generation of code templates containing all essential interfaces – based on UML diagram (e.g. with XMI)
Implementation

- Reuse of prefabricated components or
- Implementation of new components according to specifications
  - Modern tools support automatic generation of code fragments including:
    - inheritance relationships
    - interface declarations
    - constructors & method bodies
  - Developer must only fill in application logic
- No consideration to distribution specific aspects

Implementation corresponding to component model e.g. OSGi, EJB, .NET, but independent of a concrete vendor-specific platform
### Deployment

- Installation of implemented components within a concrete runtime environment (application server)
  - Component extension to include specified configuration
  - Code generation
  - Creation of additional objects and/or components

- Configuration of distribution specific component behaviour
  - Transaction context, security aspects, persistence, component, etc.
  - Separation from application logic
  - Deployment Descriptor: attribute declaration using XML based notation

- Tools for definition and modification of system configuration
Middleware: Application Server - Architecture

Java RMI, AJAX, SOAP

Java Client

HTTP

Outer Firewall

Web-Server

HTML Documents

CGI Scripts (optional)

Inner Firewall

Application Server

proprietary Protocols

Transaction Monitors

proprietary Protocols

Business software

proprietary Protocols

Mainframe applications

Stateful-connection

Stateless-connection

Data bases

HTTP

Transaction Monitors

HTTP

Mainframe applications

Java RMI, AJAX, SOAP
Multi-tier architecture

Client access to application server with standardised protocols
- Java RMI with Java-based Application Server
- SOAP and web services with all other approaches (e.g., .Net)

Web server is point of entry
- Comfortable initial access by means of HTTP

Firewalls protect server side
- Outer Firewall: access rights (based on IP addresses and TCP ports)
- Inner Firewall: authentication and authorisation at the user level

Examples:
- Java EE: Oracle, IBM, and many others
- Microsoft .NET
  - Strongly oriented towards Windows platform → limited portability and compatibility
- Open Source: JOnAS, WildFly, Apache TomEE etc.
Tasks & Characteristics

- Support for at least one component model
- Access to database transactions - including main products (e.g. IBM DB2, Oracle)
- Provisioning of security mechanisms
- Automatic replication of application server on middle tier \(\rightarrow\) load balancing
- High scalability (several dozen servers \(\rightarrow\) several 1000 clients)
  - Massive employment of threads
  - Optimisation through caching, replication and clustering
- Enterprise Application Integration (EAI) and legacy software integration
  - Provisioning of interfaces and tools for integrating existing backend applications
  - Simple case: EAI achieved through data integration or interface adaptation
  - Complex cases: EAI achieved through workflow based integration
- Integration of vendor-specific development environments
- Support of Java APIs (JDBC, JNDI, JMS, Web etc.)
Instantiation

- Creation of instances of deployed components during runtime
  - Instance creation, activation, deactivation
  - Synchronization of persistent data with backend

- Container controls component lifecycles
  - Instance creation, activation, deactivation
  - Synchronization of persistent data with backend

- Runtime environment tools to monitor the system
  - Control of replication and clustering of components and/or distributed servers
  - Management of distributed transactions
  - Role definition for access control to system resources

- Testing and Debugging – extended requirements for distributed systems
Goal of debugging: Fault elimination to ensure error-free software execution

- Fault finding during testing
- Control and inspection of internal program runtime
- Interaction interface to the System Under Test (SUT) to control program flow

Requirements:

- Symbolic debugging (String c = “xyz” instead of “LOC FF2243 AC32...”)
- Reproducibility (quasi-deterministic)
- Presentation of state information – (Variables, Registers, Ports etc: “show c”)
- Modification of system state – (set c = “ABC”)
Special Requirements & Problems

- Extended functionality (distributed components and remote communication)
- Intervention at message exchange level
- Concurrently active and parallel threads and processes
- Absence of a global state and common clock
- Semantics of special constructs (breakpoint, break conditions)
- Indeterminism (original execution to be reproduced during testing)
- Interference of debugger and system
- Information flooding (high number of system states, huge state information)
Debugging: Inter-Process Communication

- State information contains in addition to process-/object state also communication state ⇒ direct manipulation required for testing
- Separation into intra-process layer (conventional) and inter-process-layer (distributed)
- Debugger functionality of the inter-process layer
  - Message manipulation:
    - insert <m> in <port>
    - read <m> from <port>
    - extract <m> from <port>
    - forward <m> to <port>
  - Break points
    - set break <port> <mtype> [send | receive]
    - set break <port1> ... <portn>
- Statistic accounting records
  - e.g. number of invocations, periods of blocking etc
- Access to operating system objects (Semaphores, Processes)
Problem: no common clock and storage
⇒ no consistent state representation

Approaches
- Clock synchronization - limited accuracy (in range of milliseconds)
- Logical arrangement of the events – Lamport Approach

Lamport-Approach
- Partial-order ("→") “Predecessor-Relation”
- Events are ordered by causal context
- event a before event b
- Unordered if events are independent
Rules

• a and b in the same process, a before b : a → b
• a to send, b to receive a message : a → b
• (sending before receiving)

• a → b, b → c ⇒ a → c (transitive)

⇒ All essential events for distributed processing can be ordered (consistent logical “snapshots”)

⇒ Reproducibility of program flows for testing and debugging
Realization of the algorithm

- Each process has local event counter \( Z \) (initially “Null”)
- Each intra-process event has a number \( N(E) \), as well as each message send \( N(S) \) and receive \( N(R) \) event

- Intra-process Event:
  - \( Z := Z + 1 \)
  - \( N(E) := Z \)

- Sending:
  - Increment of \( Z \) \( (Z := Z + 1) \)
  - Mark message sending event: \( N(S) := Z \)
  - Send this counter together with the message to the receiver

- Receiving of message with number \( N(S) \)
  - If \( N(S) > Z \) (Receiver) set \( Z := N(S) + 1 \)
  - otherwise set \( Z := Z + 1 \)
  - Receiving event counter \( N(R) := Z \)

\[ \text{-> causally related events are in increasing order} \]
• Causally related events ordered completely
• Non-causal events ⇒ unordered (e.g. (P2.12, P3.12) or (P1.7, P2.9))
### Debugging: Consistent Breakpoints

- **Problem:**
  - Time delay after issuing of a halt-command

- **Approach:**
  - Backtracking to consistent state directly before a stopping event ("reset line")

- **Procedure:**
  - Backtracking of the causal contexts regarding to the predecessor-relation of messages
  - Minimum requirement: send events of received messages must be part of the breakpoint (e.g. \( t_{23} \) related to \( t_{12} \), or \( t_{32} \) related to \( t_{21} \))

![Diagram](image)

- Process 1
- \( t_{11} \)
- Process 2
- \( t_{21} \), \( t_{22} \), \( t_{23} \), \( t_{24} \)
- \( t_{12} \): stop point event
- Process 2: Backtrack to \( t_{23} \)
- Process 3
- \( t_{31} \), \( t_{32} \), \( t_{33} \), \( t_{34} \)
- Process 3: Backtrack to \( t_{32} \)
Debugging: Handling Indeterminism

- Indeterministic program behavior: Race conditions
  - E.g. Online Shop - Two customers order same product but there is only one of this product left in stock. Outcome depends on “race” between messages

- Solutions:
  - Testing of different possible execution sequences via distributed Single Step
  - Re-execution / Replay via output recording based on Lamport approach

- Approaches:
  - Re-execution (whole system)
    - recording of all inter-process events
    - control of repeated execution based on this
    - sequence of events checked against original run
    - Significant overhead for re-starting the whole system
  - Replay (single process and inter-process events)
    - also involves recording of all inter-process events, but together with their data
    - replay of only a single process possible by feeding in its recorded inputs (important also for technical/engineering processes with peripheral devices)
    - Simpler due to limitation to one process, but more data-intensive
Problem:
- Large number of processes and remote communication
- Large volume of information

Requirements:
- Recorded / output information to be reduced
  - Filtering of information for particular processes/message types
  - Targeting only inter-process events and only relevant time intervals
- Visualization of information
  - Control windows
  - Animation tools
- Abstraction forms for
  - Groups of interacting process
  - Execution (Timing-Diagram)
  - Ports (abstract message flow)
• Modified Requirements
  o Changing business processes
  o Extended usage possibilities expected by customer
• Experiences gained throughout lifecycle fed back into requirements analysis \(\rightarrow\) refined software design
• Creative approach necessary - hardly automatable
Summary

- Component approach offers:
  - High level of abstraction
  - Separation of configuration from implementation
  - Extensive reuse
  - Simplified development through UML based modelling and code generation
  - Configuration of distribution specifics at deployment, followed by debugging

- Traditional object-oriented and message oriented solutions still appropriate for simple applications, encapsulated into component-based approaches

- Tool support available throughout software lifecycle