8. Development of Component Based Distributed Systems

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

Component-based software design of distributed systems
- Requirements
- UML-based Specification
- Deployment based on Application Servers
- Instantiation and Monitoring at runtime
- Distributed Debugging
Software Design & Lifecycle

- **Distributed Systems – Characteristics:**
  - Long lifespan
  - Subject to constant change

- **Resulting design & lifecycle requirements:**
  - Readily changeable
  - Maintainable
  - Extendable

- **Tool support – required throughout lifecycle for:**
  - System modelling
  - Automation of routine tasks -> code generation
  - Gradual refinement – from domain specific design to:
    - concrete component models
    - platform technologies
    - executable component instances
  - Remove burden of distribution specific aspects from developer
Software Components – Views

Four Views on Component according to Cheeseman and Daniels:

- **Component Specification**
  - “Casing” of a component; independent of its implementation
  - Specification of component’s offered and required interfaces
  - Interfaces specified separately from other component parts
  - Architecture independent from individual implementations

- **Component Implementation**
  - Implementation of a component according to a given specification
  - Definition of the “content” of the “casing”
  - Different implementations can belong to a particular specification
Software Components – Views (cont.)

- **Installed Component (Deployment)**
  - Installation of component implementations on a component platform
  - Registration of implementation with component platform
  - Configuration information separate from component code
    - Describes behaviour with/use of platform services by the component
    - E.g. transaction processing, persistence, security

- **Component Object**
  - Component in runtime
  - Representation of an instance of a component
  - Possession of an identity
  - Encapsulation of application logic and state information
Software Design & Lifecycle – Stages

Requirements Analysis → Specification → Generation / Implementation → Deployment

- Requirements Analysis
- Specification
- Generation / Implementation
- Deployment

Component-Specification → UML (Unified Modelling Language) → Component Implementation → Installed Components

Modified Requirements → Specification

UML → EJB / .NET / OSGi

Component Implementation

EJB Container, .NET, OSGi

Executed Component Instances

Instantiation

Installed Components
Requirements analysis

- Specification of structural and functional aspects:
  - User orientated concepts and terminology
    - Easy to understand
    - Abstraction from technical details
  - Readily changeable
  - Readily maintainable
  - Feasible to implement
### 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 → shopping basket, product catalogue etc.
- Modelling with UML
Application Modelling with UML

- Unified Modelling Language – UML
  - Based on object orientated 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
Hierarchical Modelling

Order Processing component composed of further components. (See next slide)
Application Modelling with UML

- **Hierarchic Modelling – Component out of components**

  ![Diagram](image)

  **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 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

- **Purpose**
  - Realization of application functionality on server side
  - Interface-server between Web/Java-Client and services of enterprise data processing ("middle-tier")

- **Tasks & Characteristics**
  - Support for at least one component model
  - Access to databases - including main products (e.g. IBM DB2, Oracle)
  - Realization of distributed transaction protocols
  - Provision of security mechanisms
  - Automatic replication of application server on middle tier  
    → load balancing
  - High scalability (several dozen servers -> several 1000 clients)
    - Massive employment of threads
    - Optimisation through caching, replication and clustering
Middleware: Application Server

- **Tasks & Characteristics (continued)**
  - Enterprise Application Integration (EAI)
    - Provision of interfaces and tools for EAI
    - Simple case: EAI achieved through data- or call adaptation (data integration or interface integration)
    - Complex cases: EAI achieved through Workflow based integration
  - Legacy-integration
  - Binding to server applications in background
  - Integration of development environment
    - E.g. IBM WebSphere Studio, Oracle Weblogic Workshop, MS Visual C#, IBM Rational Rose, ArcStyler etc.
  - Support of current Java APIs (JDBC, JNDI, JMS etc.)
  - Support of WWW-services (e.g. installation of HTML, Servlets etc.)
Middleware: Application Server - Architecture

- **Java RMI, AJAX, SOAP**

- **HTML-Client**
  - HTTP
  - Outer Firewall

- **Web-Server**
  - HTML-Documents
  - CGI-Scripts (optional)

- **Application-Server**
  - Transaction-Monitors
  - Business software
  - Mainframe-applications

- **Data bases**
  - Stateful-connection
  - Stateless-connection

- **Outer Firewall**
  - HTTP

- **Inner Firewall**
Middleware: Application Server - Architecture

- 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 Weblogic, IBM WebSphere
  - Microsoft .NET
    - Strongly orientated towards windows platform → limited portability and compatibility
  - Open Source: JOnAS, Jboss Application Server, etc.
Software Design & Lifecycle - Stages

- **Instantiation**
  - Creation of instances of deployed components during runtime
  - Container controls component lifecycles
    - Instance creation, activation, deactivation
    - Reconciliation 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

(UML (Unified Modelling Language))

EJB / .NET / OSGi

EJB Container, .NET, OSGi

Instantiation
Aim: ensure software operates error free in standard conditions

Method: Debugger

• Fault finding during testing
• Control and inspection of internal program runtime
• Interaction interface to the System Under Test (SUT)
• Enables developer to exert targeted influence on program flow

Requirements for Debugger:

• User-friendliness
• Problem-orientation (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”)
• Supervision mechanisms
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 (additional work to be reproduced in testing)
- Interference “Debugger ⇔ System”
- Monitoring deadlocks and reset-procedures during distributed transactions
- Identification and remediation of scalability bottleneck
- Resulting information flooding (high number of system states, large data volume for 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:
    o insert <m> in <port>
    o read <m> from <port>
    o extract <m> from <port>
    o forward <m> to <port>
  - Break points
    o set break <port> <mtype> [send | receive]
    o set break <port1> ... <portn>
- Statistic accounting records
  - e.g. number of invocations, periods of blocking etc
- Access to operating system objects (Semaphore, Processes)
Debugging: Consistent State Representation

- 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
Debugging: Lamport Approach

- **Rules**
  - a and b in the same process, a before b: \( a \rightarrow b \)
  - a to send, b to receive a message: \( a \rightarrow b \)
  - (sending before receiving)
  - \( a \rightarrow b, b \rightarrow c \Rightarrow a \rightarrow c \) (transitively)

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

\( \Rightarrow \) Reproducibility of programme 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 the messages ($\delta = N(E)$)

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

Sending:
- increment of $Z$ ($Z := Z + 1$)
- mark Sending Event: $N(E) := Z$
- mark message: $\delta := Z$

Receiving of message with number $\delta$
- if $\delta > Z$ (Receiver) set $Z := \delta + 1$
- otherwise set $Z := Z + 1$
- Receiving Event $N(E) := Z$

i.e. counter at receiver should be incremented so it is more than counter at sender

If $E \sigma \rightarrow E \rho \quad N(E) < N(E) \rho$
• Causal events ordered completely
• Non-causal events $\Rightarrow$ unordered (e.g. Nr.12 within P2 and P3)
• Events ordered to same time point can be distinguished by prefixing process name e.g. P2.12 and P3.12
Breakpoints:
- Pausing programme at specific locations
- Examination of individual process through step by step execution of instructions
- Global breakpoint required for a distributed system

Problem: when does a break point satisfy distributed conditions?

Approach:
- Predicates determine the conditions or state at which a process should be stopped (mathematical relations or boolean functions)
- Simple predicates (one process, “call proc”)
- Disjunctive predicates (“P1: call proc | P2: call xy”)
- Subjunctive predicates (“P1: call proc & P1: x=1”) allowed only inside one process
- Joint predicates: coupling of events in predecessor-relation:

```
Process 1
\[ t_{11} \]
\[ t_{12} \]

Process 2
\[ t_{21} \]
\[ t_{22} \]
\[ t_{23} \]

Process 3
\[ t_{31} \]
\[ t_{32} \]
\[ t_{33} \]
```

\[ t_{31}, t_{22} : \text{ordered} \]
\[ t_{11}, t_{21} : \text{unordered} \]
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: sender(s) of received message(s) must be identifiable in breakpoint (i.e. send event is part of the breakpoint)

---

**Diagram:**

- Process 1:
  - t₁₁
  - t₁₂
  - t₁₃
  - t₁₄

- Process 2:
  - t₂₁
  - t₂₂
  - t₂₃
  - t₂₄
  - t₁₂: stop point event
  - Process 2: Backtrack to t₂₃

- Process 3:
  - t₃₁
  - t₃₂
  - t₃₃
  - t₃₄
  - Process 3: Backtrack to t₃₂
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
  - use Lamport approach to record all events

- **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
    - High storage requirements but reduction via checkpoint with full status information
  - Replay (single process and inter-process events)
    - Also involves recording of all inter-process events
    - Replay of only a single process possible by feeding in its recorded inputs (important also for technical/engineering processes with peripheral devices)
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 inter-process events
  - Targeting relevant time intervals
- Visualization of information
  - Control windows
  - Animation tools
- Abstraction forms for
  - Groups of interacting process
  - Execution (Timing-Diagram)
  - Ports (abstract message flow)
Debugging: Architecture Proposal

Centralized dialogue process

Computer A
- local debugging control
  - Process 1
  - Process 2

Computer B
- local debugging control
  - Process 3
  - Process 4
• Modified Requirements
  o Changing business processes
  o Extended use possibilities expected by customer
• Experiences gained throughout lifecycle fed back into requirements analysis → refined software design
• Exacting and creative approach necessary - hardly automatable
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

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

Tool support available throughout software lifecycle
References