Component-based Distributed Systems

Thomas Springer
thomas.springer@tu-dresden.de
https://www.rn.inf.tu-dresden.de/springer
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

- Characteristics
  - long lifespan
  - Subject to constant change

- Resulting design & lifecycle requirements
  - readily changeable
  - maintainable
  - extensible

- Tool support – required throughout lifecycle
  - 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
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
- **installed Component (Deployment)**
  - Installation of component implementations on component platform
  - Registration of implementation with component platform
  - Configuration information separate from component code
    - describes component’s behaviour with / use of platform services
    - 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
    - UML (Unified Modelling Language)
    - Component Specification
    - Modified Requirements
  - Generation / Implementation
    - Component Implementation
  - Deployment
    - EJB Container, .NET, OSGi
    - Installed Components
    - Instantiation
    - Executed Component Instances

Schill, Springer, Hara
Distributed Systems – Component-based Distributed Systems
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
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. (cf. 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 description (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: XML-based attribute declaration notation

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

- **Purpose**
  - Realisation 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)
  - Realisation of distributed transaction protocols
  - Provision of security mechanisms
  - Automatic replication of application server on middle tier → load balancing
  - High scalability (several dozen servers → several thousand clients)
    - Massive employment of threads
    - Optimisation through caching, replication and clustering
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 by workflow-based integration
- Legacy-integration
- Binding to server applications in background
- Integration of development environment
  - (e.g., IBM WebSphere Studio, Oracle Weblogic Workshop)
- Support of current Java APIs (JDBC, JNDI, JMS, etc.)
- Support of WWW-services (e.g., installation of HTML, Servlets)
Middleware: Application Server - Architecture

- **Java RMI, AJAX, SOAP**

- **Java-Client**
- **HTML-Client**
- **Web-Server**
  - HTML-Documents
  - CGI-Scripts (optional)
- **Application-Server**
  - Transaction-Monitors
  - Business software
  - Mainframe-applications

- **Outer Firewall**
- **Inner Firewall**

- **Data bases**

- **HTTP**
  - Stateless-connection
  - Stateful-connection

- **Proprietary Protocols**

- **Distributed Systems – Component-based Distributed Systems**
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 through HTTP

Firewalls protect server side
- outer Firewall: access rights (based on IPs 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
- Open Source: JOnAS, Jboss Application Server, etc.
**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
- **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) (\texttt{String \texttt{c}="xyz" instead of LOC FF2243 AC32...})
- Reproducibility (quasi-deterministic)
- Presentation of state information (variables, registers, ports, etc.: \texttt{show c})
- Modification of system state (\texttt{set c="ABC"})
- Supervision mechanisms
Debugging: Requirements for DSs

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 resets during distributed transactions
- Identification and remediation of scalability bottleneck
- resulting information flooding (high number of system states, large data volume for state information)
- in addition to process/object state, state information also contains 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., # of invocations, periods of blocking)

- Access to operating system objects (Semaphore, processes)
Problem: no common clock and storage; thus, 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
- let \(a\) and \(b\) in the same process, \(a\) before \(b\) : \(a \rightarrow b\)
- let \(a\) to send, \(b\) to receive a message : \(a \rightarrow b\)
- (sending occurs before receiving)
- transitivity: let \(a \rightarrow b\) and \(b \rightarrow c\) : \(a \rightarrow c\)

⇒ all essential events for distributed processing can be ordered (consistent logical ‘snapshots’)
⇒ Reproducibility of programme flows for testing and debugging
- each process has a local event counter $Z$ (initially null)
- each intra-process event has a number $N(E)$, as well as each message a send ($N(S)$) and receive ($N(R)$) event
- Intra-process Event:
  - $Z := Z + 1$
  - $N(E) := Z$
- Sending:
  - $Z := Z + 1$
  - $N(S) := Z$ (mark message sending event)
  - send this counter together with the message to the receiver
- Receiving of message with number $N(S)$
  - if $N(S) > Z$ set $Z := N(S) + 1$
  - otherwise set $Z := Z + 1$
  - $N(R) := Z$

causally related events are in increasing order
Debugging: Lamport Approach

- causally related events ordered completely
- non-causal events $\rightarrow$ unordered (e.g., (P2.12, P3.12) or (P1.7, P2.9))
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 \(t_{21}\))
indeterministic programme 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
Debugging: Handling Indeterminism

Approaches

- Re-execution (whole system)
  - recording of all inter-process events based on Lamport-approach
  - control of repeated execution based on this
  - Sequence of events checked against original run
  - high storage requirements but reduction via check points with full status information

- 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)
- **Problem**
  - large number of processes and remote communication
  - large volume of information

- **Requirements**
  - recorded information to be reduced
    - filtering of information for particular processes/message types
    - targeting inter-process events
    - targeting relevant time intervals
  - visualisation 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**

- changing business processes
- 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 UML-based modelling and code generation
- Configuration of distribution specifics at deployment

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

Tool support available throughout software lifecycle

