Software Architecture for Adaptive Systems

Sam Guinea
DEI – Politecnico di Milano
Milano – 20th November 2013
Self-adaptive software

- Defined by Laddaga in the 1997 DARPA Broad Agency Announcement as:
  - “…software that evaluates its own performance and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do…”
Self-adaptive software

- Aims to adjust various artifacts and attributes in response to changes in
  - The **self**, that is, the whole body of the software, usually implemented in several layers
  - The **context**, that is, everything in the operating environment that affects the system’s properties and its behavior
- Its lifecycle cannot be stopped after its development and initial setup
Self-* properties

- Self-adaptiveness
  - Self-configuring
  - Self-optimizing
- Self-healing
  - Self-protecting
- Self-awareness
  - Context-awareness
Open vs. closed systems

• Closed systems
  – Fixed set of elements
  – Adaptation can only act on them to keep the system on track

• Open systems
  – Elements can appear and disappear
  – Adaptation must both “discover” existing elements and act on them to keep the system on track
Anticipated vs. un-anticipated

- Anticipated adaption (closed)
  - Situations to be accommodated at run-time are known at design-time

- Un-anticipated adaption (open)
  - Possibilities are recognized and computed at run-time
  - Decisions are computed by using self-awareness and environmental context information
Externalized adaptation

- One or more models of the system are maintained at runtime
  - They are used to identify and resolve problems

- Changes are described as operations on the model

- Changes to the model affect changes onto the underlying system
Different needs

- **Topology**
  - Different interactions among the same elements
  - New elements enter the system
- **Behavior**
  - Same elements start behaving differently
  - New elements are injected in the system
- **Control**
  - MAPE elements must be added
  - Reliability and robustness must be enforced
Software Architecture

• Key part of software design
  – Not all design is software architecture
• It focuses on the topology of the system and also on
  – Quality attributes (e.g., performance)
  – Non-functional requirements (e.g., cost)
• Architecture provides an abstract view of a system
Some definitions (I)

- ANSI/IEEE Std 1471-2000
  - Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution

- Software Engineering Institute
  - The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them
Some definitions (II)

- Garlan and Shaw
  - [Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the overall system **structure** emerges as a new kind of problem. Structural issues include **gross organization** and **global control structure**; **protocols for communication, synchronization, and data access**; assignment of functionality to design elements; **physical distribution**; **composition** of design elements; **scaling** and **performance**; and selection among design alternatives.
Architecture (summary)

• Architecting a system means structuring it

• Architecture defines a comprehensive view of the system where components are clearly identified

• Architecture defines:
  – Component interfaces
  – Component communications and dependencies
  – Component responsibilities
Components and connectors

• Components
  – Encapsulate related functions
  – Encapsulate related data

• Connectors
  – Model interactions among components
  – Separate computation from interaction
  – Minimize component interdependencies
  – Support software evolution
Structure

• Too many dependencies are bad!

• Key issues
  – Identify components that may change
  – Reduce direct dependencies on these components

• Focus on modifiable systems
Architectural styles

• Catalogue of successfully used structures
• Have well-known characteristics appropriate for different requirements
• Are very useful as:
  – Reusable architectural blueprints
  – Efficient communication means

• Large systems comprise a number of individual styles/patterns
Architectural views

- A software architecture represents a complex design artifact that requires many possible views:
  - **Logical view** describes architecturally significant elements of the architecture and the relationships between them
  - **Process view** describes the concurrency and communications elements of an architecture
  - **Physical view** depicts how the major processes and components are mapped on to the applications hardware.
  - **Development view** captures the internal organization of the software components as held in e.g. a configuration management tool
  - **Architecture use cases** capture the requirements for the architecture; related to more than one particular view

Three key concepts

• Architecture
  – Single instantiation of a given solution
  – Special- vs. general purpose solution

• Style
  – Reusable assembly of components and connectors
  – Usually domain-agnostic solution
  – High-level interaction patterns

• Framework
  – A universal, reusable software platform that provides generic functionality
Styles vs. technologies

- These technologies
  - Provide applications with fundamental services for distributed computing
  - Insulate applications from underlying platform
Adaptability (I)

- Components
  - Give each component a single, clearly defined purpose
  - Minimize component interdependencies
  - Avoid burdening components with interaction responsibilities
  - Separate processing from data
  - Separate data from meta-data
Adaptability (II)

• Connectors
  – Give each connector a clearly defined responsibility
  – Make the connector flexible
  – Support Connector Composability
  – Be aware of differences between direct and indirect dependencies
• Configurations
  – Leverage explicit connectors
  – Try to make distribution transparent
Some architectural styles
Pipe and Filter

• Components are filters that transform input data streams into output data streams
• Filters are independent and have no knowledge of the others
• Components are connected through pipes
Blackboard

- Central data structure (blackboard)
- Components operate on the blackboard
- System control is entirely driven by the blackboard state
Rule-based

- Inference engine parses inputs
  - If it is a fact, it adds this entry to the knowledge base
  - If it is a query, it searches the knowledge base for applicable rules and attempts to resolve the query
- Behavior can be very easily modified through rules
- Understanding the interactions among multiple rules can become very difficult
Publish-Subscribe

- Subscribers register to receive specific messages or specific content
- Publishers produce messages
- One-way dissemination of information with low-coupling of components
Service-oriented

• Business functionality is grouped into self contained and reusable units called services.
• Services are autonomous and discrete units of functionality that are usually accessed remotely.
• Services can be evolved independently, moved, and scaled at runtime.
REST

- REST was coined by Roy Fielding in his Ph.D. dissertation to describe a design pattern for implementing networked systems

- Every distinguishable entity is a resource.
  - A resource may be a Web site, an HTML page, an XML document, a Web service, a physical device, etc.

- Every resource is uniquely identified by a URL

- All interactions between a client and a web service are done with simple operations
  - Most web interactions are done using HTTP (GET, PUT, POST, DELETE)
A look at research
Phase 1

Software Architecture as a Design-time tool for Systems that Need to be Adaptive

1. Weaves

- Architectural Style with accompanying notation
- Pipe and Filter with three differences
  - Tool fragments process object streams (NO byte streams)
  - Connectors are explicitly sized queues
  - Tool fragments can have multiple inputs and outputs

1. Weaves

- **Ports and Queues**
  - Increases the flexibility of interconnectivity
  - Allows for Location transparency
  - Specialized ports help solve output -> input incompatibilities
  - A full queue sends an error to the sender, which waits and tries again

- **Tool Fragments**
  - Lifecycle management: start, suspend, resume, sleep, abort

- **Analysis**
  - Self-metric tool fragments
  - Instruments (specialized tool fragments) inserted in the weaving
  - Observers (separation between data capture and analysis)
2. C2

- Obtain the benefits of MVC in a distributed and heterogeneous setting
- Layered network of concurrent components hooked together by explicit message-based connectors

2. C2

- Benefits
  - Substrate independence
  - Accommodating heterogeneity
  - Can easily support component substitution
  - Design in a MVC style
  - Support for concurrent components
  - Support for network-distributed systems
  - Smart connectors can support filtering policies

3. Darwin and Regis

- **Configuration Programming**
  - Separate the description of the program structure from the programming of the functional components and communication
  - Manage growing complexity
- Based on the notion of **provided** and **required** interfaces

```
Component sensornet (int n){
  provide sensin <port smsg>;
  require sensout <port smsg>;

  array P[n]:poller;
  inst
  M: mux;
  D: demux;

  forall i:0..n-1 {
    inst P[i] @i+1;
    bind
    P[i].output -- M.input[i];
    D.output[i] – P[i].input;
  }

  bind
  M.output – sensout;
  sensin – D.input;
}
```
3. Darwin and Regis

- Hierarchical configuration allows for a scalable solution

- Regis provides C++ automatically generated templates for implementing communication and processing components

- Allows for Dynamic Configuration meaning the system’s structure can change over time
  - Dynamic instantiation
  - Lazy instantiation

Magee, Dulay, and Kramer, “Regis: a Constructive Development Environment for Distributed Programs”, Distributed Systems Engineering, Volume 1, Number 5, 1994
4. The Need for Self-Adaptation

- UAVs are used to disable an enemy airfield
  - Midway, intelligence finds that SAMs are defending the airspace
  - Autonomous re-planning leads to two groups of UAVs (a SAM-suppression unit and a airfield suppression unit)
  - This leads to the automatic deploy of new SAM recognition algorithms

- Components are added to fielded and heterogeneous systems with no downtime

- This article defines what a Self-Adaptive System is and needs

- Required assurances: consistency, correctness, and distributed change coordination
4. Reasoning on Self-Adaptive Software

What conditions?

Open- or closed-adaptation?

Type of autonomy?

Frequencies?

Cost-effectiveness?

Information type and accuracy?

Evolutionary Programming (algorithm generation, genetic algorithms, AI-based learning)

Generic or parameterized algorithms

Online algorithms (deterministic, randomized, or probabilistic)

Conditional expressions

Algorithm selection

4. The Proposed Methodology

Phase 2

Software Architecture for Self-Adaptive Systems

2001 – **Dowling et al.**

2003 – **Gomaa et al.**
“Dynamic Software Reconfiguration in Software Product Families”

2003-2004 – **Garlan et al.**
“Increasing System Dependability through Architecture-based Self-repair”
&
“Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”

2004-2005 – **Hawthorne et al.**
&
“Architectural Styles for Adaptable Self-Healing Dependable Systems”

2007 – **Kramer and Magee**
“Self-Managed Systems: An Architectural Challenge”
1. Software Product Families

• Software Product Family
  – A Software Architecture that characterizes the similarities and variations that are allowed among the members of a product “family”.

• Software Configuration
  – Process of adapting the architecture of the product family to create the architecture of a specific product member

• The paper discusses **Dynamic System Reconfiguration**
  – Non interference with the parts that are not affected
  – Components should complete their activities prior to reconfiguration
  – Separation of Reconfiguration and Application concerns

1. Reconfigurable Evolutionary Product Family Life Cycle

Reconfigurable Product Family Specification, Reconfigurable Product Family Architecture, Reconfigurable Component Types, Reconfiguration Patterns

Product Family Engineering

Product Family Reuse Library

Target System Configuration

Executable Target System

Target System Reconfiguration

Product Family Requirements

Software Patterns

Unsatisfied Requirements, Errors, Adaptationa

Reconfiguration Requirements

1. Reconfiguration Patterns

- How do components cooperate to change their configuration using reconfiguration commands?
  - This is tied to the software patterns we adopt

- Each component has
  - An operating statechart
    - operational transactions
  - A main reconfiguration statechart
    - explains how the component passes through active, passivating, passive, and quiescent states during reconfiguration
  - One or more operating with reconfiguration statecharts
    - For handling reconfiguration events in the operating statechart
  - One or more neighbor component state tracking statecharts

1. Decentralized Control System
Reconfiguration Pattern
1. Change Management Model

- Change Rules
- A component can only be removed if quiescent
- Interconnections can be unlinked if the component is quiescent with respect to those links
- Etc.

- Change Transaction Model
  - Impacted Sets (of components)
    - Components that must be brought to quiescence
    - Behavioral, Topological, Inter-component Behavioral, and Reconfiguration Pattern Impacted
  - Reconfiguration Commands
    - Passivate, checkpoint, unlink, remove, create, link, activate, restore, reactivate

2. Architecture-based Self-Repair

- Provides a generalization of architecture-based self-adaptation
- The architectural style becomes a **first-class run-time entity**
- The style determines
  - What needs to be monitored
  - What constraints need to be evaluated
  - What to do when there is a violation
  - How to perform the repair
- Augment the style with
  - Style-specific architectural operators
  - Collection of repair strategies

2. Architecture-based Self-Repair

- The generic model comprises **Components** and **Connectors** with explicit interfaces
  - **Ports** are component interfaces
  - **Roles** are connector interfaces
- Components can be further refined through **representations** (various levels of abstraction)
- Semantic properties are described through graph annotation (property lists)

- Style as a system of types, plus a set of rules and constraints
  - Style types are defined using Acme, a generic ADL
  - Style constraints are defined in Armani, first-order predicate logic

- A **repair strategy** determines a problem’s cause and how to fix it
- Defined as a transactional sequence of **tactics**
  - plus a policy for solving tactic conflict
- A tactic has a **pre-condition** and a **repair script**
2. A Web-based Server-Client System

Family PerformanceClientServerFam extends ClientServerFam with {
  Component Type PAClientT extends ClientT with {
    Properties {
      Requests : sequence <any>;
      ResponseTime : float;
      ServiceTime : float;
    }
  }
}

ComponentType PAServerT extends ServerT with {...}

Connector Type PALinkT extends LinkT with {
  Properties {
    DelayTime : float;
  }
}

Component Type PAServerGroupT extends ServerGroupT with {
  Properties {
    Replication : int <<default : int = 1;>>;
    Requests : sequence <any>;
    ResponseTime : float;
    ServiceTime : float;
    AvgLoad : float;
  }
  Invariant AvgLoad > minLoad;
}

Role Type PAClientRoleT extends ClientRoleT with {
  Property averageLatency : float;
  Invariant averageLatency < maxLatency;
}

Family ClientServerFam = {
  Component Type ClientT = {...};
  Component Type ServerT = {...};
  Component Type ServerGroupT = {...};
}

Role Type ClientRoleT = {...};
Role Type ServerRoleT = {...};

Connector Type LinkT = {
  invariant size(select r : role in Self.Roles | declaresType(r, ServerRoleT)) == 1;
  invariant size(select r : role in Self.Roles | declaresType(r, ClientRoleT)) >= 1;
  Role ClientRole1 : ClientRoleT;
  Role ServerRole : ServerRoleT;
}
2. Operations and Strategies

- Adaptation Operations
  - addServer()
  - move(to:serverGrouptT)
  - remove()

- Strategies
  - fixLatency
    - Pre-condition: averageLatency is NOT less or equal to maxLatency
    - Tactic 1: server group is overloaded -> create a new server in the group
    - Tactic 2: there is communication delay -> find the best server group and move the client-server connector to that group
3. An Architectural Challenge

- Architecture provides the required level abstraction and generality to deal with Self-management
  - Can help with scalability
  - Build on existing work
  - Potential for an integrated approach

- The goal is to minimize the degree of explicit management necessary for construction and subsequent evolution whilst preserving the architectural properties implied by its specification

- They propose a three-layer architecture based on Gat’s three layer architecture for self-managing robots

3. An Architectural Challenge

- **Component Control**
  - How can we preserve safe application operation during change?
  - How can we ensure safety properties are never violated?

- **Change Management**
  - How can we deal with distribution and decentralization?
  - How can we preserve global consistency and guarantee local autonomy?

- **Goal Management**
  - How can we achieve goal specification that is both comprehensible and machine readable?
  - How can we decompose goals and generate operationalized plans?
Phase 3

Ongoing Research on Software Architecture for Self-Adaptive Systems

2009–**Weyns et al.**
&
“Patterns of Delegate MAS”

2009–**Georgas et al.**
“Using Architectural Models to Manage and Visualize Runtime Adaptation”

2012–**Cheng et al.**
“Stitch: A Language for Architecture-Based Self-Adaptation”

2011-2012–**Baresi et al.**
“A-3: an Architectural Style for Coordinating Distributed Components”
&
“Coordination of Distributed Systems through Self-Organizing Group Topologies”
1. Situated Multi-Agent Systems

- The system is structured as interacting autonomous entities that are situated in an environment
  - They employ the environment to share information and coordinate their behavior
  - Control is decentralized
  - Self-management is the system’s capability to manage dynamism and change autonomously

1. Situated Multi-Agent Systems

- **Perception** – a filtered sensing of the environment
- **Decision Making** – action selection through the influence-reaction model
- **Communication** – communicative interaction with other agents

- Communication and decision making are kept separate
  - Clear separation of concerns
  - Both functions can act in parallel and proceed at different paces
1. Situated Multi-Agent Systems

- Horizontal Decomposition
  - Representation Generator
  - Interaction
  - Communication Mediation

- Vertical Decomposition
  - Observation and Data Processing
  - Low-level Control
  - Communication Service
  - Synchronization and Data Processing
1. Patterns for MAS

• Problems:
  – Global-to-local and local-to-global information dissemination
    • Information is needed for decision making
  – Stability
    • Decisions need continuous revision due to new possibilities or problems

• Common Solutions:
  – Smart Messages
    • Information retrieval is delegated to smart messages
  – Inertia Management
    • Pheromones loose value over time
1. Solutions

- Smart messages mix state and behavior
  - Behavior is executed at every node to determine:
    - How it interacts with the node
    - How it will “move” from there
    - If message cloning is required

- Delegate MAS
  - About effectively using a conglomerate of smart messages to solve the problem of repeated interactions
    - Specification of interactions, frequency of interactions, and aggregation and processing of the interaction results
  - It is a behavior module that includes
    - A policy for creating smart messages with their own parameterized behaviors and initial states
1. Solutions

- Delegate Ant MAS
  - A distributed system needs to find a suitable route through the network to allocate a task
  - The solution is similar to distributed Ant Colony Optimization
  - A Delegate Ant MAS manages multiple ant agents
    - Ant agent: smart message with an ACO algorithm
    - A single ant traverses the environment on search of a solution and reports back
    - Migration is determined by a *probabilistic rule* that takes into account pheromone values associated with connections to neighboring nodes
    - Ant agents influence the pheromone values on nodes as part of the solution (forward and backward influence)
2. Management and Visualization

- An Operations Control Center to
  - Contextualize current and past behavior with respect to the system configurations that resulted in these behaviors
  - Support retroactive analyses of historical information about a system’s composition and behavior
  - Connect to operator-driven proactive management of the system

2. Management and Visualization

- The main result is a historical graph of architectural configurations
  - **Visibility** – to see what happened
  - **Understandability** – to improve adaptation
  - **Management** – to rollback or push the system into an existing configuration

- Directed cyclic graph $G=(N, E)$
  - $N$ is a set of nodes
  - $E$ is a set of unidirectional edges between nodes
  - Each $n$ in $N$ defines a specific architectural configuration
  - The tool provides bidirectional **diffs** for each edge in the graph
3. Stitch

- A language for defining and automating the execution of adaptation strategies in an architecture-based self-adaptation framework

- Requirements for Stitch
  - Adaptation decision processes should be able to choose the next action depending on the outcome of previous ones
  - When evaluating the result of an adaptation action the language should take into account that effects could be susceptible to delay
  - Strategies should be “guarded” by activation conditions
  - Should be possible to determine the best strategy to execute if there is more than one – this must depend on the context of execution
  - Past successes or failures to adapt should contribute to the overall process

- Stitch defines repair decision trees, together with business objectives to guide the strategy selection
3. Stitch

- **Operator** - primitive unit of adaptation
  - The operators are determined by the architectural style

- **Tactic** – an abstraction that packages operators into larger units of change. A tactic contains:
  - A sequence of operator calls
  - Activation pre-conditions
  - A definition of the effects that it is attempting to achieve
  - An impact vector that specifies how it will impact the system’s quality dimensions

```java
tactic switchToTextualMode () {
    condition {
        exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;
    }
    action {
        svrs = { select s : T.ServerT | ! s.isTextualMode }; 
        for (T.ServerT s : svrs) { Sys.setTextualMode(s, true); }
    }
    effect {
        forall c:T.ClientT in M.components | c.expRspTime ≤ M.MAX_RSPTIME;
        forall s:T.ServerT in M.components | s.isTextualMode ;
    }
}
```
3. Stitch

- **Strategy** – each step is the conditional execution of a tactic
  - It is a tree of condition-action-delay decision nodes
  - Each strategy has a context-based activation condition
  - Allows for the calculation of an aggregate utility function

```plaintext
declare boolean styleApplies = Model.hasType(M,"ClientT")/.."ServerT";
declare boolean cViolation = exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;

strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
  declare boolean hiLatency = exists k:T.HttpConnT in M.connectors | k.latency > M.MAX_LATENCY;
declare boolean hiLoad = exists s:T.ServerT in M.components | s.load > M.MAX_UTIL;

  t1: (#Pr[t1] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
    t1a: (success) -> done ;
  }

  t2: (#Pr[t2] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t2a: (!hiLoad) -> done ;
    t2b: (!success) -> do [1] t1 ;
  }

  t3: (default) -> fail;
}
```

3. Stitch

- **Strategy Selection** – chooses the strategy with the highest utility. This is achieved through the definition of:
  - Quality dimensions
    - Identifier, label, description, mapping to architectural property, utility function definition
  - Utility preferences
    - Used to define business priorities over quality dimensions
  - Impact vectors
    - Costs and benefits that a tactic has on each quality dimension, defined as deltas
  - Branch probabilities
    - Captures the fact that a tactic could not achieve its effect, or that a tactic’s activation condition might not be met

The A-3 Initiative
The Problem

• High-volume and highly volatile distributed systems
  – have very strong coordination requirements, and
  – are hard to design.
  – need to be flexible to adapt to frequent changes in the execution environment or in the system’s available resources.

Think Smart Spaces

• We need to coordinate the behaviours of multiple elements so that they reach a common goal.
The A-3 solution

• A **group** allows multiple elements to be treated as a single less-dynamic entity

• A **supervisor** is responsible for coordinating its followers

• A **follower** joins groups to better understand how to behave depending on the situation at hand
Group Composition

• A system is assembled by composing groups

• Groups can be composed in many different ways
  • nodes can belong to more than one group at a time
  • nodes can play different roles in different groups

• Composition of groups leads to interacting distributed control loops
A Multi-layered Middleware

**Application Layer**
- consists of *logical groups*
- each *logical group* has only one supervisor
- each *element* can belong to more than one *logical group*

**Coordination Layer**
- each *logical group* has one hierarchical composition of coordination groups
- each *element* for each *logical group* can play different roles in the coordination groups

L. Baresi, S. Guinea, P. Saeedi “Self-managing Overlays for Infrastructure-less Networks” Proc. 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems
## Layer Evolution

### Application Layer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image9" alt="Diagram" /></td>
<td><img src="image10" alt="Diagram" /></td>
<td><img src="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
</tbody>
</table>

### Coordination Layer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image13" alt="Diagram" /></td>
<td><img src="image14" alt="Diagram" /></td>
<td><img src="image15" alt="Diagram" /></td>
<td><img src="image16" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image17" alt="Diagram" /></td>
<td><img src="image18" alt="Diagram" /></td>
<td><img src="image19" alt="Diagram" /></td>
<td><img src="image20" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image21" alt="Diagram" /></td>
<td><img src="image22" alt="Diagram" /></td>
<td><img src="image23" alt="Diagram" /></td>
<td><img src="image24" alt="Diagram" /></td>
</tr>
</tbody>
</table>
A Self-Adaptive Middleware

- **Congestion management**
  - Supervisor moves its extra followers to other sub-groups.

- **Coordination topology contraction**
  - Supervisor moves its followers to other sub-groups and delete its sub-group.

- **Coordination group supervisor failure**
  - the corresponding coordination sub-group gets disconnected from the coordination topology.
Advanced Group Capabilities

- **Group-based views**
  - to optimize the membership knowledge and improve system robustness

- **Group-based state management**
  - to store application and coordination relevant data directly in the group, and to retrieve them when needed.

*Implemented with the a Distributed Tuple-Space*
System Robustness

Supervisors are not points of failure!

• Supervisors periodically store *important application data* regarding their groups

• Followers can launch a *leader-election algorithm* to choose a new supervisor

• The stored data can be easily *retrieved*, with minimum delay, by a substitute supervisor
An Smart Hospital Example
The Example’s Design
Conclusion
Software Architecture

• Software Architecture provides a level of abstraction that allows us to
  • Scale our understanding of complex systems, and
  • Manage adaptation (and the self-* properties of our systems)

• Programming in the large vs. programming in the small

• A plethora of styles for many different system requirements
  • Yet, it is still possible to generalize and understand what helps make a system adaptable, or self-adaptable
Research in Software Architecture

• Three different phases
  • Phase 1 - Software Architecture as a Design-time tool for Systems that Need to be Adaptive
  • Phase 2 - Software Architecture for Self-Adaptive Systems
  • Phase 3 - Ongoing Research on Software Architecture for Self-Adaptive Systems

• Architecture as a living entity
  • Supporting adaptation framework
Outlook for Research

- Increasing unpredictability -> DSPLs
- Need for Reference Architectures for self-* properties
- Architectural Abstractions for Adaptation and Context
- Quality Assessment of configurations and adaptations
  - Design time vs. Deployment time vs. Run time

### Quality Assessment

- Dynamic software product lines
- Increasing level of knowledge integration
- Architectural Abstractions for Adaptation and Context
- Design time vs. Deployment time vs. Run time

**Reference Systems and Data**