Architectural Style Requirements for Self-Healing Systems

Marija Mikic-Rakic, Nikunj Mehta, Nenad Medvidovic

Computer Science Department University of Southern California Los Angeles, CA 90089-0781 USA {marija,mehta,neno}@usc.edu

What Does Self-Healing Mean to Us?

- Self-healing systems
 - Ability to adapt (e.g., reconfigure) in response to the:
 - Changes within the system
 - Changes in the execution environment
 - System faults

Targeted Applications

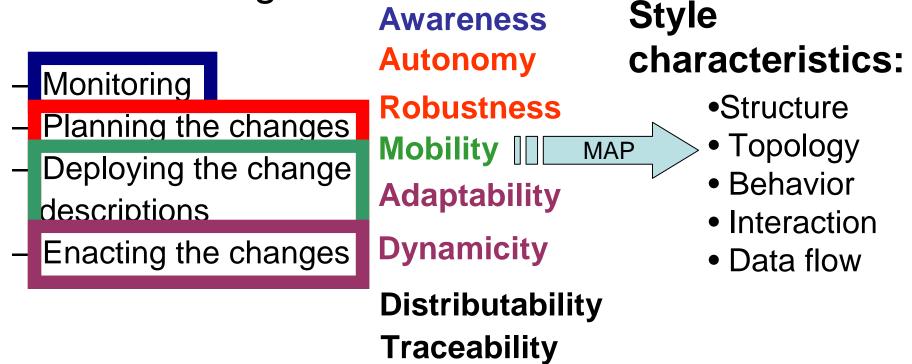
- Highly distributed
- Highly mobile
- Decentralized
- Resource constrained

Problem

- No general understanding of what constitutes an effective self-healing style
- How to evaluate or compare different selfhealing styles
- Assessing the suitability of an existing style for the self-healing domain

Approach

 Identifying architectural style requirements for self healing



of the style equirement activities)	Structure	Тороlоду	Behavior	Interaction	Data flow
Adaptability (enacting changes)	Separable components	Portals attached to a single component or connector	Exposed via named services	Asynchronous coordination	Discrete events
	Explicit connectors	onstrained number of component portals - limited component veneration of component veneration of the component ven		Implicit invocation	
	Explicit entry/exit portals	Expandable (number of) connector portals		Event-based interaction	Data streams
	Primitive ducts	Limited component dependencies (visibility)		Adjustable connector delivery policies	
Dynamicity (enacting changes)	Separable components	Components and connectors dynamically created	State preservation/restoration	Different interaction categories (e.g.,	
	Explicit connectors	Modifiable portal-to-portal bindings (i.e. ducts)	Component quiescence	allowed, deferred, disallowed)	Dynamic change events
	Explicit entry/exit portals	Dynamically expandable (number of) connector portals	Dynamism change analysis	Delivery guarantees (at least once,	System architectural mod used to analyze the validi proposed changes
	Primitive ducts	Less constrained topological rules for attaching dynamism effectors application architecture (to add the possibility of more direct contro over the architecture)		exactly once)	
	Dynamism effectors		Data queueing and buffering by connectors	Synchronous, possibly real-time, meta-level dynamic change requests	
	Dynamic change analysis agents		Dynamism effecting functionality		State transfer events
Awareness (monitoring)	Introspection portals	Less constrained topological rules for attaching to introspection facilities, to enable a more direct control over the architecture	Self-monitoring and assessment functionality	Real-time system monitoring data delivery (to monitor correctness) 🗸	System monitoring events
	Introspection interfaces		Execution trace capture	Asynchronous system monitoring	
	Meta-level components	Direct binding of monitors to components, connectors, portals, or ducts	Execution trace analysis	data delivery (to monitor statistical performance)	Critical system monitoring event patterns
	Meta-level connectors				.
	Self-monitors	Less constrained topological rules for attaching to environment facilities	Environment-monitoring and assessment functionality	Ongoing or intermittent environment monitoring	Environment-level events
	System monitors		Environment event analysis		Critical environment- level event patterns
Autonomy (planning, deploying, enacting changes)	Autonomous meta-level	Meta level components connected to dynamism effectors and change analysis components	 Adjustable planning policies 	dynamism (to ensure that the	Dynamic change events
	components				System architectural mod
	Explicit, autonomous connectors				State transfer events
Robustness (planning, deploying, enacting changes)	Autonomous components 🛛 🗸	Connectivity only via (known) portals and ducts	Exception handling Data queueing/buffering	Asynchronous coordination \checkmark	Discrete events
	Explicit entry/exit portals			Implicit invocation	
	Primitive ducts			Event-based interaction	Exception propagation
Distributability (general requirement)	Autonomous components 🛛 🗸	Distributed topology rules same as local topology rules	Data caching by distributed 🗸	Remote procedure calls (RPC)	Byte streams
	Explicit and distributed			Discrete event-based interaction 🗸	Discrete events
	connectors V		Connection setup and teardown	Continuous stream-based interaction	Ouality of interaction
	Explicit portals to remote environments			Data marshalling and unmarshalling by distributed connectors	
	Distribution channels (ducts) 🗸		Multi-tasking mechanisms such as threads		
	Distributed node registries				
Mobility (deploying, enacting changes)	Separable components	Modifiable portal-to-duct bindings	State transfer	Different interaction categories during migration (e.g., allowed, deferred, disallowed)	Meta-level mobility reque
	Distributed connectors	Limited component dependencies (visibility)	,		
	Explicit portals to remote environments	Dynamically expandable (number of) connector portals	Component quiescence		Data tuples
	Modifiable portal-to-duct bindings	Less constrained topological rules for attaching mobility effectors to application architecture	Data queueing and buffering by 🗸 connectors	Delivery guarantees (at least once, exactly once)	System components
	Mobility effectors	Mobility effectors attached to analysis agents		Data rerouting to new destinations	System architectural mod

Problems Outside of Our Scope

- Programming language support for – Exception handling
- Artificial intelligence
- Adaptive components
- Algorithms for system adaptation