Autonomic Systems

- **Autonomic**: adaptive
  - Self-healing:
    - cluster systems via node restart
  - Self-optimizing:
    - variable encoding schemes for web audio streaming services
  - Self-regulating :
    - apache web server periodically kills child processes
- **Maintenance**:
  - expensive, time-consuming

    I want my availability, but I won't do it myself
- **Automated maintenance**:
  - Cheaper
  - Quicker response than human
  - 24/7 watch, can afford to "forget and leave running"

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**

- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions

- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Build large-scale systems with reusable components

- **Inherent problem with the development of large-scale systems**
  - Hugely complex, unwise for one group of developers to create the whole thing from scratch
  - Outsource sub-projects to experts vs. license their technology
  - Integrate with COTS components:
    - Cheaper than to re-implement them
- **Software engineering and practicality reasons**
  - component has already been implemented
  - available immediately
  - no duplication of effort
  - 3 types of software components:
    - COTS
    - In-house
    - One-use, specific-purpose component

Component-based Software Engineering

- **Software component**:
  - unit of software that conforms to a component model
    - e.g. COM+, JavaBeans
  - Defines standards:
    - Composition: how components are composed together
    - Interaction: IDL description of interface elements

- **Two stages of CBSE**
  1. Component development
     - No feedback from customer
     - No waterfall model with iterations
     - Exhibit openness, adaptability,
  2. Integrating component into applications
     - Requirements analysis
     - Choose component with required functionality

Take it or leave it ...

but then go on looking for another implementation

Component-based Software Engineering – ii

- **Imperfect match in functionality and requirements**
  - "Fixed" contract
    - No means for component evolution
  - Active Interfaces [12]
    - Adaptation interface. Open policies
    - Static adaptation of component functionality
  - Interface Incompatibilities
    - Granularity of operations and data-types, interaction mechanisms, implementation languages
    - Component wrappers
    - Connectors [14]
    - SWIG, JNI, `popen(..)`, `system(..)`

- **Considerations**
  - Application builder is **not** going to re-implement the component
  - Want to maintain encapsulation, information hiding

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**

- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions

- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Static modeling of possible runtime reconfigurations

- **Runtime adaptation of software**
  - Ever-changing resource availability
  - Dynamic execution environment

- **Separation of concerns**:
  - application logic vs. adaptation

- **Granularity of adaptation**
  - Micro-level:
    - component developer-enabled mechanism, setting switches via Active Interfaces [12, 13, 16]
  - Medium-level:
    - change how components interact with the system, modify the interface [13, 14]
  - Macro-level:
    - phase in/out (groups of) components as part of the dynamic adaptation [13, 14]

Static modeling of possible runtime reconfigurations – ii

- **Self-contained adaptation within component**
  - Automatic generation of adaptation code
    - Compiler and language support for high-level specification of adaptation mechanism [13]
  - Pre-packaged adaptation mechanism [16]
- **Automatic integration of new component versions**
  - Configuration management [15]
    - Installations, updates, un-installations
  - Tentative use of new versions [14]
    - Transparent testing in deployed environment

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**
- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions
- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Writing code to implement dynamic adaptations

- **Hard to dynamically adapt components**
  - Lack proper understanding of the internals
  - Execute (un) trusted, unfamiliar code, with no idea how to fix if things fail
- **<u>Recognize</u> the need to adapt**
- **Utilize the available runtime mechanisms**
  - Pre-existing reconfiguration mechanisms
    - Dispatch directives to carry out local micro-adaptations
  - Use adaptability of middleware to effectively carry out medium- and macro-scale adaptations
  - Architectural design-driven adapted, guided by component-interaction specifications

The inability to reconfigure when required, is a form of failure

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**

- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions

- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Self-healing systems

- **Failure is inevitable**: [20]
  - human error:
    - stress level proportional to probability of making a mistake [22]
    - can shield from user error, systems lack protection from administrator's errors [22]
  - unanticipated problem:
    - beyond careful and thorough testing
    - directed security attack
    - lack of handling mechanism
  - software aging: transient bugs
    - recovery requires a restart
    - build-up of transient bugs
    - failure-prone state during execution

Self-healing systems – ii

- **Availability of system**
  - Highly resilient
    - Programmed to handle every expected problem
    - Self-heals: manages to survive unexpected situations
  - Availability ratio: MTTF / (MTTF+MTTR)
    - increase base longevity period (BLP)
    - decrease recovery time

- **Problem-handling mechanism**:
  - reactive, failure-driven:
    - detect occurred failure, follow with restart of affected subsystems from a stable state
  - preventive/proactive, failure-avoidance:
    - detect increased likelihood of failure, and gradual degradation of performance, avert imminent failure

Technique: Software Rejuvenation [18, 19]

- **Graceful termination, Immediate restart**
  - Restart at a clean, internal state
  - Build-up of transient bugs
  - Numerical accumulation errors, unreleased system resources, memory leak, data corruption

- **Levels of rejuvenation**
  - Total rejuvenation
    - Scheduled downtime can be fairly cheap
    - Minimal interruption during low usage periods
  - Partial rejuvenation
    - Transparently rejuvenate selected subcomponents
    - Decoupling between subcomponents
    - Reduced recovery time only for subsystem restart
  - Recursive rejuvenation [21]
    - Rejuvenate progressively larger subsystems recursively
    - Functional or data dependencies between subcomponents

Other self-healing techniques

- **Program check-pointing**
  - Periodically save program state to persistent storage
  - Can rewind to previous states
    - auditing, logs
    - recovery to a valid state
    - install corrective patch, resume [22]
  - The power of hindsight to enable retroactive repair
  - Demonstrates "*what if*" semantics
  - Database systems:
    - rollback to consistent state if cannot commit safely

- **Zero-tolerance of system compromise**
  - Pre-emptive defense against security attacks
    - Randomized, but valid binary code sequence
    - Sanity checking of control structures
    - Choose immediate shutdown rather than have system get compromised
  - Immediate restart, with new randomized code

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**
- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions
- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Dynamic profiling, generation of runtime data

- **Adaptation subsystem**:
  - Monitoring logic and decision-making
  - Execution of adaptation mechanism
- **Automated decision and implementation**
  - Adaptation for recovery or otherwise, without human intervention
- **Runtime model of the system architecture**
  - Decision based on evolving model
  - Runtime data generated by each component
    - Embedded probes: PSL
    - Static-adaptable Active Interfaces [12]
  - Context-dependent data format and content
    - E-mail management system: size, frequency, sender/recipient addresses, types of attachments, encryption strength

Communication of runtime data to decision logic

- **Extended RPC-style communication**
  - Client communicates with server at unknown location
  - RPC clients (execution logic) should be unaware of the presence of RPC servers (decision logic)
  - Need to multiplex emitted data
  - Asynchronous callback

    I can't wait, let me know when you're done!
  - Basic Message Passing to unknown recipients

- **Event notification system**

  Subscribe to published events-of-interest
  - Item of interest
    - Something that happened somewhere, runtime data
  - Generators of items of interest
    - Core system execution, reporting runtime data
  - Consumers of items of interest
    - Monitoring subsystem, interested in runtime data

Event systems

- **Centralized event systems**
  - event-driven GUI programming
  - Event Delegation Model: AWT, SWING, JavaBeans
    - Tightly-coupled client-server model: JINI
    - Indirection, anonymity of servers via mediator object
  - Stable execution environment
    - Well-ordered delivery mechanisms
    - Fast, reliable, predictable
- **Distributed event systems**
  - Supercharged mediator between decoupled entities
    - Filtering
    - Aggregating
    - Store-and-forward, Store-and-retrieve
    - Mutual anonymity
  - Unreliable execution environment
    - Delayed delivery
    - Data loss

Distributed event systems

- **Channel-based routing**:
  - Single channel per event type [9]

    birds of a feather flock together
  - faster turnaround time; simple, efficient delivery
  - not scalable to large classes of events
- **Subject-based routing**:
  - NNTP: events on a common theme / interest
  - Mailing lists, CVS notifications
- **Content-based (semantic) routing**:
  - Interested in a subset of a class of events
  - selective delivery via specifying acceptability criteria
  - Event-data determines propagation
  - Data replication only if necessary [10, 11]
  - Event composition [8]

Content-based event routing topologies

- **Centralized routing node**
  - Approximation of localized event system
- **Hierarchical collection of nodes**
  - Subscriptions only go up, notifications cascade down
  - Disadvantages
    - Overloading of higher-level routing nodes
    - Network partitioning via single node failure
  - Advantages
    - Simple routing algorithms
    - Simple client-server relationships amongst routing nodes
- **(A)cyclic peer-to-peer network**
  - Sophisticated routing algorithms
  - Improved fault-tolerance

Items for discussion

- **Can large-scale, distributed applications be self-healing, self-regulating, self-optimizing?**

- **Important issues with respect to automated maintenance of large-scale, software systems**
  - Harder to build. Focus on reusable components
  - Specify maintenance operations during development
  - Considering maintenance as runtime adaptations
  - Gracefully handle unfamiliar, exceptional conditions

- **Proposal: design methodology**
  - Separation of concerns:
    - Application code vs.
      adaptation mechanisms {decision logic, implementation}
  - Introspection:
    - Communicate runtime data to decision logic
  - Intercession:
    - Transport reconfiguration code from decision logic

Activation of reconfiguration code

- **Re-use events**
  - the source (client/decision logic) determines who gets reconfigured, so cannot have the server (execution logic) subscribe to these
  - event systems not designed to carry large amount of binary code, if needed for component installation, etc
- **Mobile agents [5]**

  autonomous program that executes on someone's behalf

  - decision logic instructs agents to carry out runtime reconfiguration tasks
    - Late-binding of reconfiguration mechanism at target
    - Asynchronous
    - primary advantage of agents: reconfiguration might consist of significant amount of computing, ideally performed locally at execution logic rather than a long series of RPC invocations

Mobile code infrastructures

- **Constituents**
  - Server: hosting, execution, transportation
    - Place [6]
    - Agent Server [1, 3, 7]
    - Worklet Virtual Machine: PSL
  - Agents
- **Incorporate dynamic interfaces**
  - Agent installs specific-purpose interfaces to components for customized access
  - “Wrapper while you wait”, but can configure as needed

Automatic mobility of programs

- **Strong mobility**
  - OS support for process relocation [5]
- **Weak mobility**
  - State- and code-transfer at application level
  - Programming-language, runtime support [6]
    - Special-purpose language [6]
    - Scripting languages [6]
      - Agent code is in textual form
    - General purpose language [23]
      - Late-binding of class definitions by dynamic code loading
      - Serialization of objects
  - Simulated strong mobility
    - Local function continuations [2]
    - Modified JVM [4]

Security issues: mobile code

- **A greater vulnerability: unknown code**
  - Protect agent from server, and vice versa [1, 3, 7]
- **Language support**
  - Bytecode verification in JVM
    - Type-system protection from malicious classes
    - Integrity-checking of bytecode instructions
  - Cannot define / load core system classes
- **Application-level security considerations**:
  - Authentication, authorization
    - Permissions model based on certification, credentials
  - Data encryption during transit
  - Tampering detection via digital signatures

Conclusions, future directions

- **Autonomic large-scale, distributed systems**
  - Criteria for construction and automated maintenance
  - State of the art research
    - Autonomic systems exist for specific domains
    - Technologies / tools available for building general framework for adaptation

- **Dynamic architectural modeling**
  - Accurate modeling of the system during execution
  - Decision made on evolving model
  - Adaptation heuristics based on:
    - Historical patterns
    - Temporal data

Bibliography – Mobile agents

1. **Design of the Ajanta System for Mobile Agent Programming**
   Anand R. Tripathi, Neeran M. Karnik, Tanvir Ahmed, Ram D. Singh, Arvind Prakash, Vineet Kakani, Manish K. Vora, Mukta Pathak
   Journal of Systems and Software, May 2002
2. **How to Migrate Agents**
   Matthew Hohlfeld, Bennet Yee
   Technical Report CS98-588, Computer Science and Engineering Department, University of California at San Diego, La Jolla, CA, June 1998
3. **Experiences and Future Challenges in Mobile Agent Programming**
   Anand R. Tripathi, Tanvir Ahmed, Neeran M. Karnik
   Microprocessor and Microsystems 2001
4. **Pickling threads state in the Java system**
   S. Bouchenak, D. Hagimont
   In Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS), 2000
5. **Mobile Agents: Are they a good idea?**
   Colin G. Harrison, David M. Chess, Aaron Kershenbaum
   IBM Research Report, T.J.Watson Research Center, NY, 1995
6. **Programming languages for mobile code**
   Tommy Thorn
   ACM Computing Surveys, 29(3):213-239, 1997. Also Technical Report 1083, University of Rennes IRISA
7. **Design Issues in Mobile Agent Programming Systems**
   Neeran M. Karnik, Anand R. Tripathi
   IEEE Concurrency, July-Sep 1998

Bibliography – Event systems

8. **Generic Support for Distributed Applications**
Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, Mark Spiteri
IEEE Computer, pages 68-77, March 2000
9. **Host Groups: A Multicast Extension to the Internet Protocol**
S. E. Deering, D. R. Cheriton
Network Working Group: RFC 0966
10. **State of the Art Review of Distributed Event Models**
René Meier
Dept. of Computer Science, Trinity College Dublin, Ireland, March 2000. Technical report TCD-CS-2000-16
11. **Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service**
Antonio Carzaniga, David S. Rosenblum, Alexander L. Wolf
In Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)

Bibliography – System adaptation

12. **A Model for Designing Adaptable Software Components**
George Heineman
In 22nd Annual International Computer Software and Applications Conference, pages 121--127, Vienna, Austria, August 1998. In 22nd Annual International Computer Software and Applications Conference, pages 121--127, Vienna, Austria, August 1998

13. **Language and Compiler Support for Adaptive Distributed Applications**
Vikram Adve, Vinh Vi Lam, Brian Ensink
ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001) Snowbird, Utah, June 2001 (in conjunction with PLDI2001)

14. **Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach**
Marija Rakic, Nenad Medvidovic
Proceedings of SSR '01 on 2001 Symposium on Software Reusability : Putting Software Reuse in Context

15. **A Cooperative Approach to Support Software Deployment Using the Software Dock**
Richard S. Hall, Dennis Heimbigner, Alexander L. Wolf
International Conference on Software Enginering, May 1999

16. **The Illinois GRACE Project: Global Resource Adaptation through CoopEration**
Sarita V. Adve, Albert F. Harris, Christopher J. Hughes, Douglas L. Jones, Robin H. Kravets, Klara Nahrstedt, Daniel Grobe Sachs, Ruchira Sasanka, Jayanth Srinivisan, Wanghong Yuan
In proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN) 2002

Bibliography – Dynamic healing, Miscellaneous

17. **Autonomic Computing**
Paul Horn, IBM Research

18. **Software Rejuventation: Analysis, Module and Applications**
Yennun Huang, Chandra Kintala, Nick Kolettis, N. Dudley Fulton
Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena, CA, pp. June 1995, pp. 381-390

19. **IBM director software rejuvenation**.
White paper

20. **Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies**
David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupmann, Noah Treuhaft
UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002

21. **Reducing Recovery Time in a Small Recursively Restartable System**
George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, Rakesh Gowda
Appears in Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), June 2002

22. **Rewind, Repair, Replay: Three R's to Dependability**
Aaron B. Brown, David A. Patterson
To appear in 10th ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002

23. **Dynamic Class Loading in the Java(TM) Virtual Machine**
Sheng Liang, Gilad Bracha
Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)