Lecture 1: Availability 1

Luigia Petre
Åbo Akademi University, Turku, Finland
Faculty of Technology
Department of Information Technologies
IT-Dept

- Computer Science
  - computing and programming methodologies with a focus on abstract reasoning as well as interdisciplinarity

- Computer Technology
  - an engineering programme focusing on software engineering and embedded computing

- Information Systems
  - strong business component in addition to various studies on managing software-intensive systems development.
Distributed Systems Laboratory
(Computer Science)

• Mission
  • Develop methods, techniques and tools facilitating the design of correct and dependable parallel and distributed systems

• Application areas of interest
  • Distributed algorithms, large distributed networks, multi-core architectures, control systems

• Case studies
  • Industrial software-intensive systems in areas such as communication technology, digital hydraulics, satellite software, and various control systems
Hypotheses

- Formal methods have an impact on producing quality systems and software
  - quality attributes: correctness, availability, reliability
- Formal methods have an impact on producing maintainable software
  - quality attribute: maintainability
Distributed Systems Laboratory

Leaders
- Kaisa Sere, Professor
- Elena Troubitsyna, Docent (co-leader)
- Marina Waldén, Docent (co-leader)

Senior researchers
- Linas Laibinis, Docent
- Luigia Petre, Docent

Post docs
- Petr Alexeev, PhD
- Pontus Boström, TechD
- Marta Olszewska, PhD
- Leonidas Tsiopoulos, PhD
Distributed Systems Laboratory

Current PhD students
• Fredrik Degerlund, MSc
• Kashif Javed, MSc
• Maryam Kamali, MSc
• Mats Neovius, MSc
• Sergey Ostroumov, MSc
• Inna Pereverzeva, MSc
• Yuliya Prokhorova, MSc
• Petter Sandvik, MSc
• Anton Tarasyuk, MSc
Main areas of expertise

• Formal methods
  • Research on formal frameworks for developing reliable software and systems
  • Refinement: the developed systems are correct-by-construction: a system can be described at different levels of abstraction and the consistency of and in between levels can be proved mathematically

• Dependable systems
  • Modelling safety-critical and fault tolerant systems from various domains
  • Interfacing formal models with safety analysis techniques
  • Creating patterns and process guidelines for modelling various aspects of dependability
  • Proof-based verification and model-checking of essential dependability properties
Main areas of expertise

- Control systems
  - Modelling and analysing the control systems using the **systems approach**
    - a system or a software-based product is first modeled as one entity on a very abstract level
    - detail is introduced into the model in a systematic manner and at some point, when the model is detailed enough it is decomposed into meaningful parts or sub-models, e.g. hardware and software
    - gives structure to the design of complex distributed systems and helps in managing the derivation task
  - Recent application area: developing digital hydraulic controllers, based on Simulink and Contract-Based Design

- Automated reasoning
  - Contribution to the development of the Rodin platform, a theorem prover tool for the correct development of systems in Event-B
  - Model-based verification in the Event-B model checker ProB and experiment with existing probabilistic model checkers
  - Simulation together with formal modelling based on the Simulink tool
  - Developing tool support (VERSÅA) to verify Simulink models
Main areas of expertise

- Trusted networks and services
  - Software development methods to service-oriented development of (mobile) distributed systems, peer-to-peer and sensor networks, component-based design etc.
  - Deriving fault tolerant middleware for mobile agent systems
  - Reasoning about safety and data integrity of critical multi-agent systems by refinement

- Formal hardware design
  - Developing new hardware design and verification methods for both asynchronous and synchronous circuits, system-on-chips (SoC), 2D and 3D network-on-chips (NoC), and Multi-core
Current projects

- **NODES** - Nordic Network on Dependable Systems
  - Nordforsk during 2007-2012

- **DEPLOY** - Industrial deployment of system engineering methods providing high dependability and productivity
  - EU FP7 IP project during 2008-2012

- **ASSURE** - Autonomic Software-intensive Systems: foundations of safety and resilience
  - Academy of Finland project during 2010 – 2013

- **DECO** - Formal Dependability-Explicit development model for Complex software-intensive systems
  - Academy of Finland project during 2011 – 2012

- **RECOMP** - Reduced Certification Costs for trusted Multi-core Platforms
  - ARTEMIS Joint Undertaking project during 2010-2013 financed by EU
  - Component based design methods for trusted multi-core based systems relying on contract based component models
  - Utilisation of formal methods with respect to safety critical standards
Current projects

- **eDiHy** - Energy Efficient Digital Hydraulic Hybrid Machines
  - Academy of Finland project during 2011-2014
  - Contract-based methodology for the design of control systems relying on formal software construction techniques and simulation tools
  - Building fault-tolerant platforms for reliable software

- **EFFIMA/DiHy** - Digital Microhydraulics
  - Fimecc/EFFIMA/MeKo SHOK project during 2009-2013
  - Improve controllability and fault tolerance of control systems via developing modular control code

- **EFFIMA/DigiHybrid** - Regenerative hydraulic hybrid with digi-valve and multi chamber cylinder technology
  - Fimecc/EFFIMA/MeKo SHOK project during 2011-2014
  - Prove safety and reliability of the software for the digital hydraulic technology
  - Implement a fault-tolerant, general purpose platform for digital hydraulic accumulators
Mini-series plan

- 26.4: Availability
  - Classical (measurable) availability
  - Constructive availability
  - Service availability

- 27.4: Formal methods and availability
  - Several classical formal methods
  - Action systems
  - Network availability with action systems
  - Exercises..
Dependability

The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers (IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance)

Aim

- Increasingly, individuals and organizations are developing or procuring sophisticated computing systems on whose services they need to place great reliance.
- In differing circumstances, the focus will be on differing properties of such services -- e.g., continuity, performance, real-time response, ability to avoid catastrophic failures, prevention of deliberate privacy intrusions.
- The notion of dependability enables these various concerns to be subsumed within a single conceptual framework.
- Dependability thus includes as special cases such attributes as reliability, availability, safety, security.
- The Working Group is aimed at identifying and integrating approaches, methods and techniques for specifying, designing, building, assessing, validating, operating and maintaining computer systems which should exhibit some or all of these attributes.

http://www.dependability.org/
IFIP Working Group 10.4 announced that three outstanding papers have been selected as winners of the 2012 Jean-Claude Laprie Award in Dependable Computing:

Dependability made of

• Attributes
  • Ways to assess dependability for a system
  • Availability: readiness for correct service
  • Reliability: continuity of correct service
  • Safety: absence of catastrophic consequences
  • Integrity: absence of improper system alteration
  • Security: confidentiality, availability, integrity
  • Maintainability: ability for a process to undergo modifications and repairs

• Threats
  • An understanding of the issues that can affect the dependability of a system: fault $\rightarrow$ error $\rightarrow$ failure

• Means
  • Ways to increase the dependability of a system
Threats

- **Fault**: defect in a system
  - Can lead or not to failure
  - System can have fault, but input and state conditions may cause the fault never to be executed so that an error would occur

- **Error**: discrepancy between intended and actual behavior
  - Inside system boundaries
  - Occurs at runtime: system reaches unexpected state due to the activation of a fault
  - Generated from invalid states => hard to observe without special mechanisms
Threats 2

- Failure
  - Instance in time when system displays behavior contrary to the specification
  - Fault activated $\rightarrow$ error (invalid state) $\rightarrow$ another error or failure: observable
  - Recorded at system boundary
  - Are errors that have propagated to the system boundary and became observable
  - Service fails to meet its specification
Means

- Prevention
  - Formal methods
- Tolerance
  - System still delivers (degraded) service
- Removal
  - Verification
- Forecasting
  - Prediction of faults
Attributes, Threats, Means

- Attributes
  - Availability
  - Reliability
  - Safety
  - Confidentiality
  - Integrity
  - Maintainability

- Threats
  - Faults
  - Errors
  - Failures
  - Prevention
  - Tolerance
  - Removal
  - Forecasting

- Means

Dependability and Security
Availability

- Readiness for correct service
- The percentage of time when the system is operational
- Typically specified in nines notation
  - 3-nines availability corresponds to 99.9% availability
  - 5-nines availability corresponds to 99.999% availability
- **MTBF**: Mean Time Between Failures
  - the average time between failures
  - hardware module
    - MTBF for hardware modules can be obtained from the vendor for off-the-shelf hardware modules. MTBF for in-house developed hardware modules is calculated by the hardware team developing the board
- **MTTR**: Mean Time To Repair
  - The time taken to repair a failed (hardware) module
  - In an operational system, repair generally means replacing the hardware module

\[
\text{Availability} = \frac{MTBF}{MTBF + MTTR}
\]
Hardware failures
Failures

- Hardware failures
  - Design failures
  - Infant Mortality
  - Random Failures
    - Redundancy
  - Wear Out
- Software failures depend on
  - Software process
  - Software complexity
  - Size of software
  - Team experience
  - Amount of code reused
  - Testing
Downtime

- Downtime per year - more intuitive meaning of availability
- **Availability Downtime**
  - 90% (1-nine) 36.5 days/year
  - 99% (2-nines) 3.65 days/year
  - 99.9% (3-nines) 8.76 hours/year
  - 99.99% (4-nines) 52 minutes/year
  - 99.999% (5-nines) 5 minutes/year
  - 99.9999% (6-nines) 31 seconds/year!
- Availability calculation
  - [http://www.eventhelix.com/realtimemana/tr/fault-handling/system_reliability_availability.htm](http://www.eventhelix.com/realtimemana/tr/fault-handling/system_reliability_availability.htm)
Software Architecture

- Architecture
  - Old art and ancient engineering discipline
- Software
  - The industry begun in late 40s
- Software architecture
  - Much less mature than computer hardware architecture
- Common excuses
  - Software industry is *young* and *unique*
- Yet: our economy relies on *software products*
"Our society runs on Software"

- Software today allows a brother in San Jose to call a sister in St. Petersburg.
- Software today speeds the process of drug discovery, potentially curing Alzheimer's.
- Software today drives the imaging systems that allow the early detection of breast cancer and other maladies.
- Software controls the passive restraint systems and antilock breaking systems that save children's lives in automobiles every day.
- Software powers our communication and transportation technologies.
- Software allows us to peer deep within ourselves and study the human genome.
- Software allows us to explore and understand our universe.
- And, make no mistake about it, we are just getting started.

Paul Levy
What does SA address?

- Complexity of software systems increased
  - Developing software: hundreds/ thousands person-years
  - Many software systems: complex as skyscrapers
- Designing software
  - Beyond algorithms/ data structures of the computation
  - New kind of problem: **overall system structure**
Producing software systems

- Criteria have changed
  - Computer hardware improved, affordable
  - Need for software applications exploded
  - How to specify requirements for new products and implement the software quickly, cheaply
- Earliest software product on market
  - Quality?
- New criterion: *does it have a good SA, understood by stakeholders and developers?*
Software architecture

1. Provides a *design plan* of a system
   - *Blueprint*
   - Implies purpose

2. Is an *abstraction* that helps in managing the complexity of a system

- Software architects limited by
  - Lack of standardized ways to represent architecture
  - Lack of analysis methods to predict whether an *architecture* will result in an *implementation* that meets the *requirements*
SA as a design plan

- Structural plan that describes
  - The elements of a system
  - How they fit together
  - How they work together to fulfill requirements
- Used as blueprint during development process
- Used to negotiate system requirements
- Used to set expectations with
  - Customers
  - Marketing/management personnel
Design tradeoffs

• Resolving tradeoffs may lead to
  • Sacrificing some desired qualities
    • E.g., simplicity
  • Compromising some requirements
    • Renegotiating
    • Reducing portability, modifiability, etc
SA goal

• One should strive for a **good** architecture
  • When system is implemented according to the architecture, it meets its requirements and resource budget
  • It is possible to implement system according to architecture

• Not good enough when
  • Not explicit or not comprehensive or not consistent or not understandable
The ABC cycle

- SA – result of technical, business, and social influences
- A SA also influences technical, business, and social influences
  - That, subsequently, influence future SAs
- This is the ABC cycle of influences: Architecture Business Cycle
  - Organizational goals $\rightarrow$ requirements $\rightarrow$ SA $\rightarrow$ systems $\rightarrow$ future new organizational goals $\rightarrow$ …
Stakeholders

- Architectures influenced by stakeholders
  - **Management**: low cost, keep people employed
  - **Marketing**: short time to market, low cost
  - **Customer**: timely delivery, not changed often
  - **End user**: behavior, performance, security
  - **Maintenance**: modifiability
The business considerations

• Determine qualities to be part of the system’s architecture
  • Quality attributes
• Such qualities are non-functional
• Functionality
  • Basic statement of the system’s capabilities, services, behavior
  • Sometimes the only development concern
    • Systems often need to be redesigned for
      • Maintainability
      • Portability
      • Scalability
      • Speed
      • Security etc
    • Compromise
More on functionality

- Functionality and quality attributes: orthogonal
  - We therefore need to separate concerns
- Functionality: the ability of the system to do the work for which it was intended
- Functionality can be achieved by using various possible structures
  - System can exist also as a monolithic module with no structure
Architecture and quality attributes

- Quality attributes considered during
  - Design, implementation, deployment
  - No attribute is dependent only on one phase
- Architecture critical for realizing many quality attributes
  - These qualities designed and evaluated at architectural level
- Architecture does not achieve these qualities
  - They are achieved via details (implementation)
Quality attributes

- System qualities
  - Availability, modifiability, performance, security, testability, usability
- Business qualities
  - Time-to-market, etc
- Architecture qualities
  - Conceptual integrity, etc
System qualities

- Of interest to software people since 70s
- Many definitions, own communities
- Problems
  - Non-operational definitions
  - Aspects belong to which quality
  - Distinct vocabularies
Quality attribute scenarios

- Quality-attribute-specific requirement, containing
  - Source of stimulus
  - Stimulus
  - Environment
  - Artifact
  - Response
  - Response measure
QA scenario parts

- **Source of stimulus**
  - Human/computer system/other actuator generating the stimulus

- **Stimulus**
  - Condition that needs to be evaluated when arrives at the system

- **Environment**
  - The conditions the system is in (overload/running/failed etc)
QA scenario parts, 2

- **Artifact**
  - Something that is stimulated (whole system/ certain system part)
- **Response**
  - Activity undertaken upon stimulus arrival
- **Response measure**
  - Response should be measurable in some manner so that the requirement can be tested
QA scenarios

- **General QA scenarios**
  - System independent
  - Can potentially pertain to any system

- **Concrete QA scenarios**
  - Specific to the particular system under consideration
  - Same role as *use cases* for specifying functional requirements
QA Scenario Generation

- We need to generate meaningful QA requirements for a system
- Requirements gathering phase
  - Starting point, but not disciplined enough recording
- We generate concrete QA scenarios
  - First create general scenarios from tables
  - From them derive system-specific scenarios
Availability

- Readiness of usage
- Concerned with system failures and consequences
- Failure
  - Deviation from intended functional behavior
  - Observable by system users
- Failure vs fault
  - Fault: event which may cause an error
  - Error: incorrect internal system state
Availability concerns

- How system failure is detected
- How frequently system failure may occur
- What happens when a failure occurs
- How long is a system allowed to be unoperable
- When can failures occur safely
- How to prevent failures
- What kind of notifications are required when a failure occurs
Repairments and maintenance

- **Time to repair:**
  - time until failure is no longer observable
- Automatic repair
- Maintenance: scheduled downtimes
- Probability
  - Mean time to fail/(mean time to fail+mean time to repair)
## Availability general scenarios

<table>
<thead>
<tr>
<th>Source</th>
<th>Internal/external to system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stimulus</strong></td>
<td>Fault: omission, crash, timing, response</td>
</tr>
<tr>
<td><strong>Artifact</strong></td>
<td>System’s processors, communication channels, persistent storage, processes</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>Normal operation or degraded mode</td>
</tr>
<tr>
<td><strong>Response</strong></td>
<td>Detect event and record it/notify appropriate parties/disable event sources causing faults/failures/ be unavailable for an interval/ continue</td>
</tr>
<tr>
<td><strong>Response measure</strong></td>
<td>Time interval of available system, availability time, time interval of degraded mode, repair time</td>
</tr>
</tbody>
</table>
Tactics

• Qualities achieved via design decisions
• What design decisions needed for achieving a specific quality?

• Tactic
  • Design decision that influences the control of a quality attribute response

• Architectural strategy
  • Collection of tactics
Architectural style

- A package of tactics
  - Tactics can refine other tactics
    - Redundancy is refined by data redundancy, code redundancy

- Example
  - One availability tactic: introduce redundancy
  - Implication: we also need synchronization of replicas
    - To ensure the redundant copy can be used if the original fails
Availability tactics

• Failure
  • Deviation from intended functional behavior
  • Observable by system users

• Failure vs fault
  • Fault: event which may cause a failure

• Availability tactics
  • Keep faults from becoming failures
  • Make possible repairments
Availability tactics (2)

Fault → Detection
- Ping/Echo
- Heartbeat
- Exception

Fault masked

Availability
- Recovery-preparation and repair
  - Voting
  - Active redundancy
  - Passive redundancy
  - Spare
- Recovery-reintroduction
  - Shadow
  - State
  - Resynchronization
  - Rollback
- Prevention
  - Removal
  - from service
  - Transactions
  - Process
  - Monitor

Repair made
Fault detection: **Ping/Echo**

- Comp. 1 issues a "ping" to comp. 2
- Comp. 1 expects an "echo" from comp. 2
- Answer within **predefined time period**
- Usable for a group of components
  - Mutually responsible for one task
- Usable for client/server
  - Tests the server and the communication path
- Hierarchy of fault detectors improves bandwidth usage
Fault detection: **Heartbeat**

- Comp. 1 emits a "**heartbeat**" message periodically
- Comp. 2 **listens** for it
- If **heartbeat** fails
  - Comp. 1 assumed failed
  - Fault correcting comp. 3 is notified
- **Heartbeat** can also carry data
Fault detection: Exceptions

- **Fault classes:** omission, crash, timing, response
- When a fault class recognized, exception is raised
  - A fault consequently is recognized
- Exception handler
  - Executes in same process that introduced the exception
  - Typically does a semantic transformation of fault into a processable form
Error Detection and correction

Error codes add structured redundancy to data so errors can be either detected, or corrected.
Error detection

- A system that cannot guarantee that the data received is the data sent is useless
- Data can be corrupted
  - Quite likely
  - Heat, magnetism, other forms of electricity
  - Noise
- Reliable systems must have mechanisms for detecting and correcting errors
Error types

- Errors
  - Single-bit
  - Burst
Single-bit errors

- Only 1 bit of a data unit is changed
- Least likely to appear in serial transmission
- Can happen in parallel transmission
Burst errors

- 2 or more bits in the data unit are changed
- Length of burst: from 1st to last corrupted bit; in between uncorrupted bits are possible
- Likely in serial transmissions

![Diagram showing burst errors with an example of a sent and received data unit with highlighted bits that are corrupted by the burst error.]
How to detect errors?

Sender

1010000000001010101010

1011101

Generating function

Redundancy check

Receiver

Checking function

Accept

Reject

Data & redundancy check

101000000000101010101010

1011101
Types of redundancy in LANs

- Detection methods
  - VRC
  - LRC
  - CRC
Vertical Redundancy Check

- Called also parity check
- A redundant bit (the parity bit) is appended to every data unit so that the total number of 1s in the unit (including the parity bit) is even
- Most common and least expensive
- Odd number of 1s can also be used
Illustration of VRC

Sender

Even-parity generator

Data

1100001

VRC

1

Receiver

Checking function

Is total number of 1s even?
Performance of VRC

- Detects single-bit errors
- It can also detect burst errors if total number of bits changed is odd
  - Exp: 1 error, 11100101; detected, sum is wrong
  - Exp: 3 errors, 11011001; detected, sum is wrong
  - Exp: 2 errors, 11101101; not detected, sum is right!
- Error can also be in the parity bit itself
- Random errors are detected with probability $\frac{1}{2}$
Longitudinal Redundancy Check

Original data

11100111 11011101 00111001 10101001

LRC

10101010

Original data plus LRC

11100111 11011101 00111001 10101001 10101010
Performance of LRC

- Better at detecting burst errors than VRC
- There is one pattern of errors that is still elusive
  - If some bits in one data unit are damaged and the same number of bits in the same position are damaged in another data unit, then LRC does not detect error
Cyclic Redundancy Check

- Most powerful, based on binary reduction
- Predefined binary unit called the divisor
- The data unit (DU) is appended with a sequence of redundant bits (CRC remainder) so that the resulted DU is exactly divisible by the divisor
- At destination, the received DU is divided by the divisor
  - If remainder is zero, ok
More on CRC

- Required qualities of a CRC
  - To have exactly one bit less than the divisor
  - Appending it to the DU must make the resulting bit sequence divisible by divisor
- Theory and application of CRC: straightforward
- The complication: deriving the CRC
Deriving the CRC

Receiver

Sender

Zero, accept
Nonzero, reject

$00...0$

$n$ bits

$n+1$ bits

Remainder

CRC

Data

Divisor

Data

CRC

$00...0$

Divisor

$n$ bits

$n+1$ bits

Remainder

CRC

Data
CRC generator

- Uses modulo-2 division

When the leftmost bit of the remainder is zero, we must use 0000 instead of the original divisor.

Data plus extra zeros. The number of zeros is one less than the number of bits in the divisor.
CRC checker

- Uses modulo-2 division in the same way

When the leftmost bit of the remainder is zero, we must use 0000 instead of the original divisor.
Polynomials

- CRC generator typically represented as an algebraic polynomial
- This is useful
  - Short
  - Proves the concept mathematically
Standard polynomials

CRC-12: \(x^{12} + x^{11} + x^3 + x^2 + x + 1\)

CRC-16: \(x^{16} + x^{15} + x^2 + 1\)

CRC-ITU-T: \(x^{16} + x^{12} + x^5 + 1\)

CRC-32: \(x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1\)

- 12, 16, 32 → size of CRC remainders
- CRC divisor’s size is hence 13, 17, 33
CRC performance

- If CRC respects the rules mentioned then:
  - All burst errors of length equal to the polynomial’s degree are detected
  - All burst errors affecting an odd number of bits are detected
  - Burst errors of length greater than the degree of polynomials are detected with high probability
- 32-bit CRC used in Ethernet, Token Ring