Summary for Advanced Computer Systems at University of Copenhagen 2021/2022. These notes are mostly based on the lecture slides and reading material.

Fundamentals

Common Problems of Systems

- **Emergent Properties**
  properties not showing up in individual components, but when combining those components
- **Propagation of Effects**
  what looks at first to be a small disruption or a local change can have effects that reach from one end of a system to the other
- **Incommensurate Scaling**
  as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working
- **Trade-offs**
  waterbed effect: pushing down on a problem at one point causes another problem to pop up somewhere else

System technical definition:

A **system** is a set of interconnected components that has an expected behavior observed at the interface with its environment.

Divide all the things in the world into two groups:

- those under discussion (part of the system)
- those that are not part **(environment)**
- the interactions between system and its environment are the **interface** between the system and environment

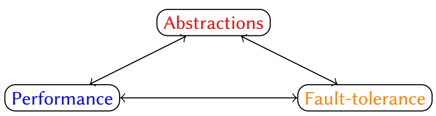
Fundamentals

- Abstractions: interpreters, memory, communication links
- Modularity with clients and services (RPC)
- Techniques for performance

Learning goals

- Identify the fundamental abstractions in computer systems
- Explain how names are used in the fundamental abstractions
- Being able to design a top-level abstraction, based on lower-level abstractions
- Discuss performance and fault-tolerance of a design

Central Trade-off: Abstractions, Performance, Fault-Tolerance



Examples for Trade-off

- To improve performance one might has to ignore the abstraction and take the behavior of the underlying concrete implementation into account
- when introducing another layer of abstraction we might introduce new kinds of errors (for example when introducing RPC, we can have communication errors)
- introducing mechanisms for fault-tolerance can have a negative effect on performance

Names

Names make connections between different the abstractions.

- Examples
  - IP-address
  - IR
- Names require a mapping scheme
- How can we map names?
  - Table lookup (e.g. Files inside directories)
  - Recursive lookup
  - Multiple lookup

Memory

- READ(name) → value
- WRITE(name, value)

Examples of Memory

- Physical memory (RAM)
- Multi-level memory hierarchy
- Address spaces and virtual memory with paging
- Key-value stores
- Database storage engines

Interpreters

Interpreter has:

- Instruction repertoire
- Environment
- Instruction pointer

Interpretation Loop:

```
do forever
  instruction <- READ(instruction_pointer)
  perform instruction in environment context
  if interrupt_signal = True
    instruction_pointer <- entry of INTERRUPT_HANDLER
    environment <- environment of INTERRUPT_HANDLER
```

Examples of Interpreters:

- Processors (CPU)
- Programming language interpreters
- Frameworks (e.g. MapReduce, Spark)
- layered programs (RPCs)

Communication links

- SEND(linkName, outgoingMessageBuffer)
- RECEIVE(linkName, incomingMessageBuffer)

Examples of Communication Links:

- Ethernet interface
- IP datagram service
- TCP sockets
- Message-Oriented Middleware (MOM)
- Multicast (e.g. CATOCS Causal and Totally-Ordered Communication System)

Other Abstractions

- Synchronization
  - Locks
  - Condition variables & monitors
- Data processing
  - Data transformations
  - Operators

System Design Principles

Design principles applicable to many areas

- **Adopt sweeping simplifications**
  So you can see what you are doing.
- **Avoid excessive generality**
  If it is good for everything it is good for nothing
- **Avoid rarely used components**
  Deterioration and corruption accumulate unnoticed - until next use.
- **Be explicit**
  Get all of the assumptions out on the table
- **Decouple modules with indirection**
  Indirection supports replaceability
- **End-to-end argument**
  The application knows best
- **Escalating complexity principle**
  Adding a feature increases complexity out of proportion
- **Incommensurate scaling rule**
  Changing a parameter by a factor of ten requires a new design
- **Keep digging principle**
  Complex systems fail for complex reasons
- **Law of diminishing returns**
  The more one improves some measure of goodness, the more effort the next improvement will require
- **Open design principle**
  Let anyone comment on the design; you need all the help you can get
- **Principle of least astonishment**
  People are part of the system. Choose interfaces that match the user's experience, expectations, and mental models
- **Robustness principle**
  Be tolerant of inputs, strict on outputs
- **Safety margin principle**
  Keep track of the distance to the edge of the cliff or you may fall over the edge
- **Unyielding foundations rule**
  It is easier to change a module than to change the modularity