CMPSCI 630 Systems **Fall 2014**

Lecture 10

Lecturer: Emery Berger *Scribes: Shaylyn Adams, Theodore Sudol*

The collection of papers discuss in this lecture all deal with system experiences and software engineering and architecture.

10.1 MIT Vs. New Jersey

The Rise of 'Worse-is-better' by Richard Gabriel is a widely cited and used paper. It describes two opposing design philosophies that are known as the **MIT** and the **New Jersey** (worse-is-better) approaches.
The ideas behind *worse-is-better* are as follows:

- Strive for simple implementation (even at the risk of complicating interface)
- Perfection is not achieved first
- "first to market wins" (although classically not true, i.e. VHS vs. BetaMax, Linux vs. Windows)

The New Jersey approach was so named because it is the home of UNIX, C, and C++ which are examples of this design view.

10.1.1 The Canonical Case: Multics versus UNIX

Multics was from MIT and General Electric and is the example of the MIT design approach. It was a groundbreaking OS with multi users, lots of API sets, lots of built in functionalities etc. Multics was formed by design-by-committee which was a main cause of added complexity. The system was supposed to be released in 1963 but it took them until 1969/70 to finish.

UNIX conversely was created by just 2 men who wanted their system to be up and running as soon as possible so they could use it. Hence, UNIX was made by the end user, design-by-user. In the process they designed the C language in order to achieve portable gaming and based it on BCPL which is simple and has one type, the byte. UNIX in the style of NJ, hid complexity via the abstraction where "everything is a file". It avoided the monolithic approach and was built out of many parts. All the basic utilities were easy to reach and right there for the user.

Obviously when looking at systems today, UNIX is the winner in this case and out survived Multics because it was simple and easier to adopt and incorporate into systems.

10.2 Cult-of-personality Languages

Many programming languages have been designed in the New Jersey style with the main goal of solving a real world problem. **Cult-of-personality languages** refers to languages designed by a *guru*, without any real regard to standard design practices.

The "terrible" language of Perl is example of this. It was created to solve the performance and portability problems with using the command line to do things like

```
\%ls | grep -v pdf | wc -l
```

So Perl (practical extraction report language) combines the ideas of grep and awk (a scripting language at command line) and has many ways to do everything. The main take away being that Perl provides tools to *solve a specific problem.* Examples of cult-of-personality languages are:

- Perl *(Larry Wall)*
- Python *(Guido van Rossum)*
- Ruby *(Yukihiro "Matz" Matsumoto)*
- PHP *(Rasmus Lerdorf, later became design-by-committee)*
- JavaScript *(Brendan Eich)*

They are often known as "curly braces style" languages which enforces the idea that they are friendly environments. They all can do useful things quickly and often interoperate with C easily. Many can be written as functional languages as well.

Why are these languages still used? Practical, easier, nicer. The environment for them has been established. Developed ecosystems and answered certain concerns people had in reality.

10.2.1 Functional Languages

Functional languages are the other kind of programming language ideology which fits under the MIT approach. These languages have mathematical foundations and often have a significant learning curve. Examples include:

- LISP
- Scheme *(made by MIT)*
- Haskell
- ML

Purely functional = program cannot see state, thus manipulation of state is not allowed.

```
a = f(x);
b = f(x);
These are the same thing.
```

Functional languages are not as popular since they were made to improve upon existing programming languages instead of solving a real world problem. They reside in the realm of academia. Haskell research is still done today, but in the context of how Haskell applies to the real world, which is something obvious to languages like PHP, C and Python. *Note: Java is the middle of these two classifications*

Scala is a quasi functional/everything, mulitparadigm language. It is better than Java and runs on a Java virtual machine. Used by Twitter as a replacement for their original Ruby backend, which could not scale to the demands on their system.

10.3 Worse-Is-Better Recap

NJ style programming languages are used more in the real world than MIT styled ones. Dropbox and Google use Python. PHP is used by Facebook who made their own just-in-time compiler for it called HipHop. Clearly, worse-is-better is not necessarily pretty but it does achieve popularity. It is important to note that Gabriel presented the worse-is-better case but he was part of the Common Lisp standardization project so believed that some sort of hybrid should be sought after.

MIT	New Jersey
Multics (took far too long)	**UNIX** (uses 1 standard interface for everything)
'MIT' Languages	**Cult of Personality**
	Plan 9*

***Plan 9** was designed by Ed Wood from Bell Labs and pushed file abstraction to its limits. Introduced /proc so you can view processes as files, captures the state and memory contents. It never took off because it was made to supplant an existing, popular system and didn't solve a problem. Some of its innovations did get absorbed into UNIX though.

Take home message: Design a pretty and simple language that explicitly solves a new/real problem. It does not necessarily have to be deep and academically satisfying.

10.4 UNIX

UNIX introduced many innovations and current standards for operating systems:

Symbolic links: Labels that point to any file or directory, target may be nonexistent. Essentially alias for target.

Hard links: Direct link to a file, must be on same volume. Essentially alias for target data.

Inode: Filesystem index, stores attributes and disk location of file

Filesystem: Data structure stored in persistent memory

Everything is a file: Directories, devices, network sockets, processes...

Piping and I/O redirection: Feed output of command into input of another, or redirect output from stdout to a file

10.5 End-to-End Argument

Saltzer and Clark presented the **end-to-end** argument which is about pushing complexity to the 'ends', i.e. the application level. This argument is demonstrated through the development of the Internet. TCP is an example and it provides reliable delivery at end and in sequence. Problems arise with *fault tolerance* and *security*.

Packet vs. circuit This distinction arose when deciding how the Internet should provide communication and tolerate some failures. **Circuits** were used with telephony and a fixed routing is established once between A and B. There is no overhead because there is no routing during communication. However, if the channel goes down, everything is lost. This is a single point of failure and there is no fault tolerance. **Packets** on the other hand, are like the mail system and practice the end-to-end policy. They contain a source, destination, TTL (time to live) and sequence number. If one route goes down or is unavailable, packets can recover by finding a new path. Once an acknowledgment signal, ACK, is received, the sender knows the message was delivered.

In the same vein as packets, UDP uses *datagrams* and it is okay if some get lost. This is applicable to things like Skype, streaming media, and phone calls. It is up to the applications to decide how to handle a lost packet. For example, Skype could replace a lost packet with just silence.

Fault Tolerance: How well a system handles errors. An example of a fault is a lost datagram or a file that was corrupted during transmission. In the end-to-end argument, the application must be able to handle the faults.

Since the Internet was made for scientists to communicate in the event of a nuclear attack (like interstate highways were 'created' to protect us against Communists), security was a low priority so there was no built in security checks. *Checksums* on whole entities instead of single packets helps to ensure reliable and correct transmission. This utilizes end-to-end checking and is simple and successful since providing reliability inside can slow things down and be superfluous.

Distributed Denial of Service Attacks

- Pretend to shop but don't actually buy anything so real customers are blocked out
- Use 'comprised' yet normal computers
- One solution is *IPSec* which makes sure packets are not modifiable

Security is slowly being integrated into the Internet and will be the 'next generation'.

Note: Lampson's hints were not thoroughly discussed; it was noted that they do actually work and have important impacts on systems