6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 1: September 13, 1993

Lecturer: Bill Weihl *Scribe: Brent Phillips and Deborah Wallach*[1]

1 Course Outline

This course is a graduate level introduction to the principles of computer systems, with a focus on concurrent and distributed systems. Instead of being based on case studies, the class will attempt to abstract general principles from the example systems. The primary tools for designing real systems are rigorous specifications and abstraction.

There will be no text, but there will be papers, handouts, problem sets, a final exam, and maybe a midterm. Grades will be based 50% on problem sets, and 50% on exams. Lecture notes for last year are available on Athena in the 6.826 locker, or from LCS Publications as Research Seminar Series #22.

The course is based on three topics:

- **Specifications and Abstraction**, which are used to abstract the visible behavior of a system from the details of its implementation.
- **Reasoning About Programs and Systems**, which is aimed at proving the correctness of programs and systems.
- **Performance**, including methods of performance enhancement and issues in design trade-offs.

These issues will be discussed in the context of the following specific areas and techniques:

- **Optimization Techniques**, such as caching.
- **Fault-Tolerance**: building systems which keep working in the face of errors.
- **Concurrency**, where several processes are allowed to run in parallel and interact with each other. This course does *not* cover a study of parallel algorithms or parallel methods for solving specific problems.
- **Distributed Systems**, where a system consists of several nodes spread across a network.
- **Other Issues**, such as security and naming.

[1] These notes are based on those of Umesh Maheshwari (September 11, 1991) and Fred Chong (September 14, 1992).

2 Principles of Computer System Design

A list of principles useful in the design of computer systems could conceivably be very long. As noted earlier, we shall focus on three factors: abstraction, reasoning techniques, and tradeoffs between performance, functionality and complexity.

Abstraction refers to the process of using specifications to separate the functionality of a system from the details of its implementation. Specifications tell *what* a system should do, without dictating *how* it should be done. In other words, a specification should be *definitional*, not *operational*. As an example, the specification for a `SORT` routine may state merely that it permutes the input sequence of numbers such that the output sequence has them in increasing order. The particular sorting algorithm used to achieve this is irrelevant. The distinction between what is and is not a part of a specification is subjective, depending on which aspects of the system's behavior its users find interesting. For instance, if the `SORT` routine is required to run in $O(n \log n)$ time, then this fact would appear in the specifications, and less efficient implementations like `InsertionSort` would be ruled out.

The use of abstractions in design has some obvious advantages:

- It provides modularity and modifiability. That is, the implementation of a module can be replaced without changing its specification, and therefore, other modules using it remain unaffected.
- It allows the designer to avoid premature optimization. For example, one could employ an inefficient but simple-and-sure implementation to begin with, and worry about performance only after the integration of the system is complete.
- It is a good way to document functionality.

Specifications can be viewed as contracts between users and implementors. The users must rely *only* on the properties in the specification. The implementors must provide those properties. Although an implementation may have certain fortuitous unspecified features, users should not rely upon them. The implementors are free to change such features in future versions of the system.

For an example of a specification, let us attempt to list the essential features of a wristwatch:

- The displayed time must be accurate to within some small delta.
- User interface—the display and input mechanisms must be of a certain format.
- The energy source must have a reasonable lifetime.
- The watch must be waterproof to a certain depth.
- Aesthetics—the watch must be under a certain weight, the strap of a certain type.
- The price must be under a certain amount.

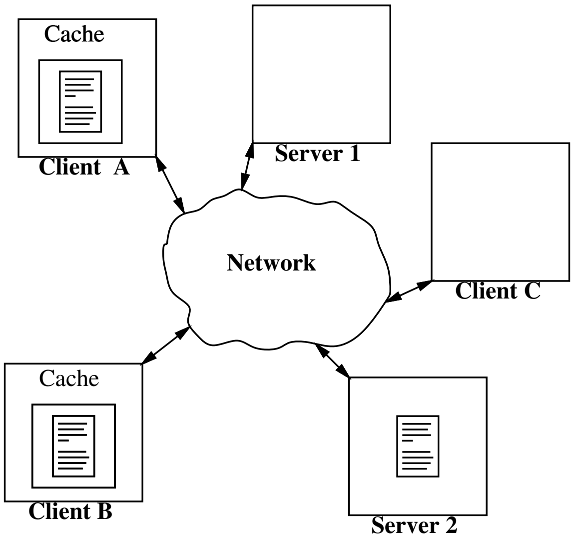

Figure 1: Performance/functionality tradeoff in a distributed system.

The actual specification a watch-designing company would use, however, depends upon user needs.

The behavior of modules is usually well-understood in a sequential world, but has more complicated meaning for concurrent and distributed systems, where events may be occurring while a process is executing. Specifications for a concurrent system are often *weaker* than those for a sequential system. This results from the non-determinacy allowed by the presence of concurrency. Forcing determinism in such a system may result in degradation of performance.

As an illustration, consider a network of file servers and clients (Figure 1). If *Client A* updates a file before *Client B* reads it, the most straightforward semantics would guarantee that the modifications are reflected in the data read by *Client B*. However, performance would improve tremendously if *Client B* could use a cached (potentially old) copy of the file. In this scenario, modifications to the file are propagated to *Client B* shortly after they occur. The extra performance requires that the semantics be relaxed; thus, there are tradeoffs between good performance and the desirable functionality with simpler specifications.

3 Writing Specifications

Specifications can be written either formally or informally. Although informal specifications are better than no specifications at all, they are often ambiguous. Formal specifications are sometimes hard to read, but are convenient to reason about.

In this course, both specifications and implementations will be written using one language: *Spec*. Spec is based on Edsger Dijkstra's *Guarded Commands*, as extended by Greg Nelson, and augmented with constructs for concurrency and atomicity by Butler Lampson.[2]

[2]Dijkstra, E., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the*

Using only one language allows successive refinement of a specification into a full-fledged implementation. On the other hand, this system has some pitfalls of its own. One can easily lapse into using the implementation style while coding the specifications, thus over-constraining the specifications. Indeed, an implementation *is* a plausible specification. Conversely, one might be tempted to use some of the succinct constructs provided for specifications while writing the implementation, which may not translate into realistic or efficient code. Thus, we need to exercise some "self-restraint" in using Spec for the two different purposes.

We now turn to what the specification for a procedure might look like. It could conceivably be a function mapping the input arguments and the initial state to the results returned and the final state. Here, the results encapsulate both the values returned normally and the exceptions raised; the state variables are used to capture side-effects. However, the use of a *function* for the specification fails to capture the non-deterministic behavior of some procedures. That is, for the same input and initial state, a number of output values or state may be acceptable. Some examples of such procedures are:

- `SQRT(x)` which might be allowed to return any value within $\sqrt{\mathtt{x}} \pm \epsilon$ for want of better precision.
- `SEARCH(item:T, s:SEQ T)` which returns any index `j` in `s`, such that `s(j)` $=$ `item`, or raises a `NotFound` exception if none exists.
- The storage allocator, which might return any of the several available free blocks on request for a chunk of memory.

Thus the specification for a procedure should be expressed as a *relation* between the input arguments plus the initial state, and the results plus the final state, allowing a one-to-many mapping. In fact, even a particular *implementation* of the procedure may be nondeterministic (e.g., if it involves concurrency), and then it too must be denoted by a relation rather than a function.

Suppose that the relation dictated by the specification is R_S and that realized by the implementation is R_I. The requirement that the implementation should satisfy the specification translates as

$$R_I \subseteq R_S$$

In words, all possible outcomes of invoking the implementation on a given set of arguments and initial state must be included in the outcomes allowed by the specification. It is possible that R_I is actually a function while R_S is not. For instance, if an implementation of the `SEARCH` routine discussed above always returns the first location (if any) of the item in the sequence, then the implementation is indeed deterministic. Implicit in the subset relation is the requirement that the domain of R_I be the same as the domain of R_S, otherwise the implementation ignores some legal inputs.

ACM., August, 1975, pp. 453–457.

Nelson, G., "A Generalization of Dijkstra's Calculus," *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989, pp. 517–561.

Neither of these references will help students learn Spec; they are provided only for historical interest.

Before winding-up, we take a look at issues of *termination*. It is possible for a procedure to not terminate at all for some inputs. We can denote this case by a special state $\perp$, pronounced *bottom*. An implementation is said to be *partially correct* if it satisfies the specification whenever it terminates. It is said to be *totally correct* if it is partially correct and always terminates.