A. Language Comparison

This section reviews other languages and systems relative to the requirements for robust servers and open distributed systems. It also compares the capabilities of Joule with those of its antecedents, Actors and concurrent constraint languages.

Joule is designed to be a foundation for distributed applications. It is largely a language foundation because a language can be layered on top of any operating system, enabling the foundation to be made extremely portable. Since the design captures many of the ideas from good operating systems, the semantics also match well with network and machine operating systems.

Many people consider language and system comparisons a soft and subjective art. The discussion of server and market-based computation in Section 1.1 provides the foundation for a hard-edged discriminant for comparing languages and systems: Can they build robust servers? This requires encapsulation, concurrency, and resource management.

The present comparison only applies the robust servers criterion. Future versions of this section will also document the large contributions from previous languages. In the present comparison, however, we consider only those features which are lacking in a particular language but present in Joule. We offer our apologies if this narrow focus makes the comparison seem invidious, and hope to amend this lack in future versions.

A.1. Language Comparison

Languages like C and C++ do not have true encapsulation; any program in the same address space can violate the modularity of objects by using casts. Systems that glue together separately-written C or C++ programs to attempt to support distributed programs are considered separately. They don't provide C or C++ any more support than they provide to assembly language programs.

Even if these programming languages provided modularity, they still lack concurrency. Again, concurrency tools like threads that are provided by the operating system are really a property of the operating system, and are considered in the next section.

The Smalltalk language and system are extremely good, but all implementations provide hooks for the debugger that any object can use to violate modularity; the language semantics provide encapsulation but

the environment throws it away. Smalltalk is also a sequential language. The standard support for concurrency is a Semaphore mechanism with non-preemptive multi-tasking; this is not sufficient for reactive, concurrent systems.

The Linda language and system can be considered together because they share the same semantics. Linda assumes a global tuple space for communication. In a distributed system, the existence of that global structure is untenable—machines fail, networks partition, etc. The reliability of a robust server cannot depend on the reliability of machines on which the robust server is not running. Further, the tuple-space of Linda is insecure: tuples are just placed in the space and other processes pattern match against the tuple-space to extract *any* tuple they recognize. Proposals have been made for M-Linda, a Linda with multiple tuple-spaces that was moving towards the Joule communication semantics, but progress on that front stopped.

Languages like Actors, ABCL/1, and Hermes, and concurrent logic programming languages like FCP, FGHC, and Janus all have the requisite encapsulation and concurrency properties. They do not have sufficient resource management capabilities to implement robust servers well, but they come the closest of existing programming languages. Many of the logic programming languages suffer from the additional disadvantage of a global environment for procedure definitions; such global constructs break in large, distributed systems. The Joule computational model is a direct combination of features from the Actors computational model and the concurrent logic programming model.

A.2. Operating Systems

Operating systems on consumer platforms (Macs and PC) often provide no encapsulation, so they cannot be a foundation for robustness. More sophisticated operating systems like Windows NT and UNIX have more encapsulation—they provide inviolable accessibility boundaries between applications—but hidden in their complexity they throw away the security in the abstractions that they provide to programmers (in UNIX, every program that runs with root permissions is a potential hole through which the entire system could become vulnerable). The resulting systems have holes that are fatal flaws when connected into a network with untrusted clients.

Micro-kernel operating systems like Mach start with capability security and provide a clean enough environment that they can be secure. They also clearly provide concurrency. The mechanisms for resource management don't provide much flexibility, but they are capable of supporting robust servers.

KeyKOS goes one step further. KeyKOS is a capability-based, secure operating system that provides hooks for explicitly managing computational resources such as processor time and memory. The only remaining lack is that it does not provide transparent forwardability of message passing between operating system objects (called Domains). As a result, the KeyKOS model cannot be extended to a network transparently.