Mobile Computation

Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Moving code, off-line and on-line

Looking back a few years, we may notice that we finally abandoned assembly language programming in almost every domain. How did that happen? In part, improvements in compiler technology and hardware speed made high-level languages competitive. But the main reason is that assembly code is inherently not portable: one cannot recompile it for a new architecture. Since recompilation is an off-line process, let us say that assembly code is *not off-line portable*. The main problems this causes are:

- *It is difficult to automatically translate assembly code to new architectures (with reasonable performance).*
- *New architectures have been emerging faster than any feasible rate of manual recoding for legacy software.*

New techniques can handle legacy assembly code, such as emulation and emulation-backed translation. But the combination of the two problems above has overwhelmed any consideration based on absolute coding efficiency. As a result, new programs are now written in off-line portable languages: they are routinely recompiled for different architectures.

We can now draw an interesting analogy. Until very recently no major language was *on-line portable*. That is, one could not take a running program and port it to a different architecture while the program was running. This, however, is precisely what must happen with network computations, because:

- *It is difficult to recompile source code on the fly for a new architecture (with reasonable performance).*
- *Connections to computers based on unknown architectures are established faster than the time it takes to recompile source code.*

Techniques have emerged to get some of the advantages of both off-line and on-line portability, such as just-in-time compilation and run-time linking. But the emphasis is now on mobility and quick compilation, not on optimized code generation.

Mobility poses a new basic question: what is the effect of taking a running computation and moving it to another network site? In most current languages, this makes little sense; the mechanisms for doing so are usually unavailable, and the effect would likely be unpredictable. In order to move computations we need languages and models where mobility makes sense; that is, where its effects are well defined.

Traditional languages and traditional compiler technology are not well suited for the world of network computing. Languages that are not off-line portable have already been abandoned (effectively, except for legacy and specialized tasks). In a similar way, languages that are not on-line portable will be abandoned because they do not provide what is increasingly perceived as basic functionality: mobility.

Moving computation, not just code

The framework we are interested in is that of *mobile computation*; that is, the notion that a computation starting at some network node may continue execution at some other network node. This framework involves much more than just moving code. Pure code mobility is useful, and has been used to great advantage in the form of Java applets, but it is also limiting. Fortunately, the other necessary components of mobile computation have already been widely studied and used.

The popular Remote Procedure Call (RPC) model is based on the notion of *control mobility*: a thread of control originating at some network node continues execution at some other network node, and then comes back. No code is moved in this process, just control, so the question of on-line code mobility does not arise.

The RPC model implements also *data mobility*: data is exchanged over the network in the form of parameters and results of RPC calls. This data must be on-line portable: data structures are *marshaled* (converted to portable form) at the originating side, sent over the network, and *unmarshaled* at the receiving site into corresponding data structures, possibly within a different computer architecture.

Some RPC systems also provide *link mobility*: the endpoint of a network connection can be sent over another network connection. The receiving party is then connected to the other endpoint.

Unlike RPC, mobile computation is based on the movement of code, not just the running of code that already exists in network nodes. The other components of RPC, however, are all very important. In mobile computation, control must move as well: the code that is transferred must be run. Data must also move, in order to preserve the state of mobile computations across moves. Network links must also move, since they are part of the state of the computation (at least, in models of mobility that support remote connections).

If code is represented as data (e.g. as the instruction stream of an interpreted virtual machine), then data mobility immediately implies code mobility. Therefore, mobile computation can be implemented rather easily over the RPC model by representing mobile code as mobile data, and taking advantage of the other facilities already provided by RPC. In fact, mobile computation can be implemented on top of various transport mechanisms, although in each case it may acquire some peculiarities of the transport. RPC is currently the most convenient substrate on which to implement mobile computation. HTTP can also be used, resulting in a more Web-oriented semantics of mobility. Thread and address space transport has been provided in the past by

some operating systems and programming languages, but usually only within a single computer architecture.

How computation moves

I wish to compare three relatively well-defined and distinct models of mobile computation. Other models certainly already exist, and more will be developed in the future. These three models differ in what kind of entities can be transmitted over the network.

The most basic form of mobility consists in just moving code; this model is represented by Tcl [3] and Java [2] (pre- Remote Method Invocation). In these languages, an architecture-independent representation of program code (source text, or bytecodes) is shipped over the network and interpreted remotely. When code moves, the current state of the computation (if any) is lost, and connections that the computation had at the originating site vanish. State and connectivity must be reestablished at the receiving site. Control is reestablished by dynamic binding or dynamic linking.

A *computation*, however, is more than just code: it is code plus the context of its execution. A computation, in this sense, can be represented as a *closure*, which is the runtime description of a running procedure. Obliq [1] takes the approach of moving closures: the code and the necessary context in which the code operates are transmitted. The context may include data, active network connections which are preserved on transmission, and new connections that are created to keep the closure in touch with the site it leaves behind. In this approach live, active, computations can move, and their meaning is preserved upon transmission. Control is reestablished by running the closure at the receiving site, possibly supplying arguments that provide local information.

Telescript [4] takes the approach of moving *agents*. Agents are similar to closures in that they carry their context with them as they move from location to location, and are reanimated at each location. Agents, however, are meant to be completely self-contained. They do not communicate remotely with other agents; rather they move to some location and communicate locally when they get there.

Obliq is in a sense the most general of these three models: a closure with no data is just code; a closure with no connections is an agent. On a local area network this level of generality is very convenient. However, the Obliq model is also the most fragile when used in full generality over the Web: the rich set of connections created by Obliq computation may be easily upset by network unreliability. In contrast, the pure code and pure agent models can at least survive intermittent connectivity failures. Even those models, though, are supplemented in practice with forms of remote connectivity, making them partially vulnerable to network instability. It is not clear yet how this tension between generality and reliability can be solved.

Fundamental issues

The very notion of mobile computation is evolving rapidly. We should expect to see the emergence of new forms of mobile computation, and of new way of using existing

mechanisms. I conclude by listing some basic questions that should be asked of any present or future mobile computation scheme.

- *What does a mobile computation do?* This is the simple issue of meaning. It has been common to extend existing implementation models to network programming with little regard for clean and consistent semantics. What are the meaningful models for mobile computation?
- *Where does computation happen?* In any model of mobility one must take the notion of multiple locations as fundamental. It should be possible to determine, in principle, where each piece of the computation happens. Location has an observable influence on behavior, on resource usage, and on the relative costs of computation versus communication. Knowing where computation happens is necessary in order to program mobile computations effectively.
- *What is the programmer's view of mobility?* There are many ready answers: distributed objects, closures, threads, continuations, agents, actors, etc. In fact, a programmer's model should be tested against the unusual realities of network programming (especially on the Web). In the long term, the prevalent models are unlikely to be exactly any of the above.
- *How is security handled?* The main obstacle to the acceptance of mobile computation for commercial applications is the issue of security, which is peculiar to code mobility. The basic technology of security is well understood, but it is not yet clear how to deploy that knowledge into languages and implementations, and how to check that security is truly respected. What is the syntax, static checking, semantics, and logic of security?

References

[1] Cardelli, L., **A language with distributed scope**. *Computing Systems,* **8**(1), 27-59. MIT Press. 1995.

[2] Gosling, J., B. Joy, and G. Steele, **The Java language specification**. Addison-Wesley. 1996.

[3] Ousterhout, J.K., **Tcl and the Tk toolkit**. Addison-Wesley. 1994.

[4] White, J.E., **Telescript technology: the foundation for the electronic marketplace**. White Paper. General Magic, Inc. 1994.