MICRO MOBILE PROGRAMS

Carl A. Gunter*
University of Pennsylvania
gunter@cis.upenn.edu

Abstract This paper describes a three-layer architecture for mobile code based on a distinction between code that is mobile versus code that is resident. We focus on code that is mobile but not resident and consider two contexts in which such code is constrained by its delivery mechanism to be small in size, resulting in *micro mobile programs.* The contexts we consider are programs carried in network communication packets and programs carried in two-dimensional barcodes.

Keywords: Mobile code, scripting language, active network, active barcode, PLAN, SwitchWare.

1. Introduction

The way computer programs are distributed is changing. Once programs came mainly from the vendor of the machine on which they ran. Open programming platforms allowed programs to be written by parties other than the vendor and possibly installed by a system administrator. Growing use of personal computers led to shrink-wrapped software, which could be bought at a store and installed by the user of a computer. Increasingly, however, programs are retrieved over the Internet by users and installed by users. This kind of installation generally takes two forms, implicit or explicit 'pull' retrieval. A user may visit a web page and implicitly download a Java applet, which is executed by the Java runtime system associated with his browser in order to provide customized functionality, like better graphics. Alternatively, a user may be told that a particular plugin is required in his browser in order to see a web page, so the user clicks on a button that instructs his browser to download and install the program that provides the desired functions.

Both of these new ways to obtain programs may be viewed as instances of *mobile* programs because their general distribution mode is to travel over the

*In Ricardo Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, pages 356–369, Montreal, Canada, August 2002. IFIP 17th World Computer Congress, Kluwer.

network. In the first case the program is *ephemeral,* because the program, an applet, is installed only temporarily while it is being used and disappears when the runtime completes running it. In the second case the program is *resident* in the form of a library like a Dynamic Link Library (DLL). There are still programs that are not really mobile, like, say, the operating system, which must generally be installed from media like a CD. We can call these *permanent* programs. Clearly there is a trend toward more minimal permanent programs in PCs. For instance, it is typical to download service packs for upgrading operating systems from web sites. Also, permanent programs are not really everlasting, since operating systems are reinstalled from CDs from time to time.

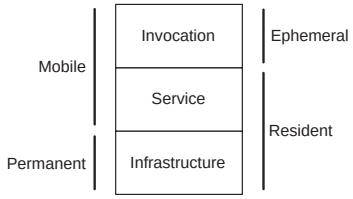

Figure 1. Three Layer Architecture

When we combine the concepts of mobile versus resident programs we obtain a three-layered architecture for mobile code as illustrated in Figure 1. The intersection of mobile and resident classes of programs yields a class of programs called *services,* consisting of programs like DLLs that are often or typically downloaded from the network. The class of permanent programs, which are resident but not mobile, forms an *infrastructure* layer. Finally, the class of ephemeral programs, which are mobile but not resident, forms an *invocation* layer. A simple example if this architecture arises with the postscript programming language [1]. A document is compiled into a postscript program. The program is sent to a printer at invocation layer to execute a printing using resident programs on the printer as libraries. After execution the ephemeral program (document) is discarded.

In this paper we consider the design issues associated with ephemeral mobile code within the three-layer architecture of Figure 1 in application contexts where the delivery mechanism for the mobile programs constrains the size of the programs in the invocation layer, resulting in what we will call *micro mobile programs.* We consider two such contexts, active packets and active barcodes. *Active networks,* as introduced in [17], allow users to program routers using active packets, which are packets that invoke custom processing functions on routers. If the programs that invoke such programs are to fit within packets, they are constrained by the packet sizes allowed by typical network path minimum transfer units, about 1300 bytes in the current Internet. *Active barcodes,* as introduced in [4], are 2D barcodes that contain computer programs. 2D barcodes provide for extremely cheap media used for delivering information in contexts like physical mail, where the barcode is printed on a letter or package. Such barcodes typically have a capacity of about 1-2 kilobytes.

The paper is divided into six sections. The second section discusses some application contexts that are used to motivate requirements and mechanisms

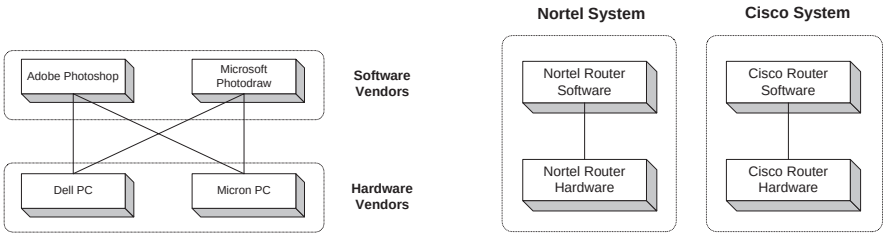

Figure 2. Horizontal Versus Vertical Programmability.

described in the third section. The fourth and fifth sections discuss active packets and active barcodes respectively. The sixth section concludes.

2. Application Contexts

A fundamental driver for software, especially for mobile code, is the issue of open Application Programming Interfaces (APIs). An open API enables third party vendors or users to write code for a platform. To see the issue in a networking context, consider IP, which assumes that communications are sent in packets that contain a small header used as data by routers. Users have little ability to program the way routers handle their packets, although facilities like ICMP and source routing provide some diagnostic and control capabilities. In the future internetworks are likely to offer more capable user customization functions such as RSVP, which enables Quality of Service (QoS) guarantees to be negotiated and allocated by routers. At a further level of programmability, routers may support the installation by owners of programs written by parties other than the router manufacturer. Consider Figure 2, which shows on the left the state of affairs for PCs, which typically provide an API usable by third party vendors. Even though this interface is generally Microsoft Windows, it is possible to run software from other vendors on top of it, resulting in a 'horizontal' software industry. By comparison, Figure 2 illustrates on the right the situation with routers, which generally provide only limited APIs for third parties, resulting in an essentially 'vertical' software industry. Developing a horizontal industry for programming routers could enable faster deployment of new functionality and more flexibility for users and owners.

Active networking concerns the idea of enabling users to run software on network elements the way one might be able to do on hosts with time-share operating systems. Research on active networking led to considerable exploration of the design options for the three-layer architecture for mobile code in Figure 1. Routers are viewed as supporting a layered model with the *NodeOS* at the infrastructure layer and a collection of *Execution Environments (EEs)* supporting various approaches to the service and invocation layers. Three examples illustrate some of the tradeoffs. The Active Network Transfer System (ANTS) [19] provides for packets that contain an identifier indicating which

of a collection of resident Java program should handle a packet received by an active router supporting the ANTS EE. If the identifier is not recognized then the host or router sending the packet is asked to send the program for this identifier so it can be installed. Thus the ANTS EE has a minimal invocation layer consisting of identifiers and a rich service layer consisting of Java programs. By contrast, SmartPackets [15] sends programs in packets that are installed on routers only as long as they execute. These consist of very short programs written in a CISC specifically designed for collecting network diagnostic information. SmartPackets therefore provides an instance of the theme of this paper, micro mobile programs.

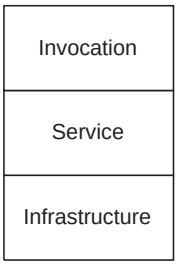

Figure 3. SwitchWare Architecture

SwitchWare [2], a third active network architecture, is depicted in Figure 3. Its infrastructure is based on the Secure Active Network Environment [3], which features secure bootstrap and remote recovery capabilities that provide for a secure and minimal permanent infrastructure. Service layer programs are written in the OCaml, a General Purpose Language (GPL). Invocation layer programs can be written in a Domain Specific Language (DSL) called the Programming Language for Active Networks (PLAN). PLAN is a scripting language whose primary construct is a remote evaluation primitive. It can be viewed as a means of composing and invoking service layer functions (written in OCaml). PLAN is a micro mobile programming language; we discuss it it more detail in Section 4 below.

Routers provide a good example of a programming application different from networked PCs, time-share servers, and web servers, but they are not the only one. Indeed, many computers are now present as *embedded* systems, that is, computers within other devices, often controlling processes these devices are involved in. A characteristic example is the software needed to control an aircraft. Embedded device programs are as diverse as the devices in which they are deployed so assumptions about networked PCs and routers are often inapplicable. In particular, embedded systems often have much different network connectivity than these other systems (especially routers). For example, airplane controller chips are likely to be connected across a networking system within the aircraft, but Internet connectivity is likely to be limited. Other programs, like those that control processes in automobiles may or may not be networked, whereas programs like the ones in a chip in a vacuum cleaner are not networked. The programs in a cell phone are an interesting example. Such programs may clearly control a communication link, but may or may not be downloadable through the network link. Another interesting dimension of embedded systems is the struggle for open APIs. Personal Digital Assistants PDAs, which are like small PCs, mostly provide an open API because PDA

vendors would like to leverage an industry of independent PDA software vendors. Automobile software is typically not programmed with an open API, at least not for users, but there is an industry of chip replacements to help users circumvent tax and environmental regulations. Cell phone software is in a middle ground with most most cell phones being programmable only by their vendors, but with substantial development of open API platforms like the Java Mobile Information Device Protocol (MIDP) providing a path to cell phones that can download applet-like programs called 'midlets' using wireless web (typically cellular) networks.

Because of complex connectivity issues and the challenge of open APIs, embedded systems provide a fertile ground for exploring new variations on the three-level mobile code architecture. This paper describes some of the progress that has been made in one specific context, that of programmable microwave ovens. Microwave ovens are familiar household and commercial cooking appliances. They often use very simple recipes, like 'cook at full power for 3-5 minutes'. Since the ovens vary from about 600 to 1000 watts in power, there is a compatibility problem that makes it necessary to provide such imprecise recipes. Moreover, recipes often involve programming the human operator as an additional actuator. As a running example consider the program in Figure 4. as studied in [4]. This is a recipe from a frozen food package. Note that the instructions 1, 3, and 4 are for the human operator and instructions 2 and 5 are for the microwave, as keyed in by the human. Recipes like this could be more sophisticated if the API of the microwave were known (eg. whether it cooks at 600 or 1000 watts) and the human did not need to key in the recipe.

```
1. Make 1 inch slit in plastic
2. 50% power for 5 minutes
3. Remove plastic overwrap
4. Rotate tray 1/2 turn
5. 100% for 1:45
```

Figure 4. Enchilada Recipe.

Open APIs for microwaves have been attempted in various forms. For instance, a pair of patents (5,812,393 and 5,883,801) provides for recipes represented with 5 to 10 digits. The idea is to put these codes on packages and have the operator key them into the microwave. The five digit codes describe the time, power level, and pause period for the device. The ten digit codes provide for two phases of cooking, similar to what we have in the enchilada recipe in Figure 4. Another idea is to put the 10 digits into a bar code and put a scanner on the microwave so it read the recipe directly from the package. This approach has a variety of limitations, especially the need to get the food vendors to put the recipes on their packages. So, another idea is to put a database of recipes into the device and look up the proper recipe based on the Universal Product Code (UPC) generally found encoded in the linear barcodes that already appear on food packages. This has the limitation that the database may become out-of-date, but this problem can be addressed by putting the device on the Internet so recipes can be downloaded. This can be done on demand, or using occasional updates the way the Tivo television sys-

tem downloads show schedules. Indeed, the Sharp Corporation demonstrated a programmable microwave at the 2000 International Housewares Show (`http://www.reviewsonline.com/IHS00.htm`) that interfaces with a PC. A descendent of this device is now being marketed in Japan. Another class of devices now approaching the market are called *multi-modal* ovens. These combine different oven technologies. A smart multi-modal device combining microwave and convection ovens was demonstrated at the 1999 International Housewares Show in Chicago (`http://www.foodtechsource.com/emag/004/gadgets.htm`). It was developed by Kit L. Yam in the Food Science Department of Rutgers University, with support from Samsung Electronics America. It reads bar codes and features a computer control with a touch screen and access to the Internet. A noteworthy aspect of multi-modal ovens is the fact that programming them is more complicated. Indeed, this complexity is a key impediment to selling them in the consumer market.

One fairly basic idea for programming a microwave to an open API is to allow recipes to be delivered using 2D barcodes. Current technology for 2D barcodes allows about 1-2 kilobytes of data to be transmitted in this way. This does not solve the problem of how to get food vendors to put recipes on packages and encounters the additional property that 2D barcodes require comparatively expensive Charged Coupled Devices (CCDs) as readers, but it is an interesting problem since it illustrates many of the issues that will arise with programming open APIs on a significant class of embedded systems. Microwaves are essentially required to have at least a rudimentary open API since food vendors are quite independent from microwave hardware vendors. Imagine the consequences of a vertical organization of this market, with Stouffer's frozen dinners that can only be cooked in a Stouffer's oven. The 2D barcode approach illustrates the idea of micro mobile programs. The programs are about the same size as those in active packets, but active barcodes are delivered by a human operator and read from a package. We discuss this in more detail in Section 5.

3. Requirements and Mechanisms

An interesting aspect of the postscript example above distinguishes it from Java applets and DLLs. Namely, the postscript program is 'pushed' to the printer. That is, the computer on which it runs is the server, not the client. In the case of Java, the client contacts the server and 'asks' for the Java applet, whereas the printer offers to the network the service of running the program provided to it by an authorized client. This distinction has various interesting security ramifications making it even more problematic in many ways than pull programs like Java and plugins already are. Mobile programming for network elements or embedded systems raises a collection of requirements of its own, and there are a variety of recurrent themes in the mechanisms that can be used to address these requirements.

3.1. Requirements

For network elements the primary challenges, as outlined in [9], are: flexibility, security, usability, and performance. Flexibility concerns just *how* programmable the network elements are. For instance, if the aim of the programmability is to allow flexible deployment of intrusion detection elements and firewalls, then there will be a need to allow the party that deploys these systems to authenticate themselves and gain access to routed packets for inspection and filtering. However, if the aim is only to provide limited diagnostic and customization features then it may be best to provide an interface that does not allow for inspection of the packets of other parties. Flexibility conflicts in general with the other requirements. Security is threatened by increased flexibility because attackers have more to work with. Even unintentional errors are more likely to cause significant harm to the network for this reason. Programmable routers are also challenged by usability in some of the same ways that microwave vendors are challenged by naive operators who must key in the recipes. Flexibility cannot appear to an endpoint as overwhelming complexity. An active network where getting a packet from its source to destination requires ingenious programming is not likely to be valuable. Performance is a key concern for active networks since custom processing times for packets must be proportionate to the benefit of custom processing. This is difficult for data path packets, so active network systems have often emphasized applications in the router control path, performing functions like configuration and diagnostics.

For embedded systems the primary challenges, as outlined in [5], are: flexibility, portability, extensibility, predictability, and deliverability. Flexibility covers quite a wide spectrum for embedded systems. In the case of microwaves it may be as simple as allowing two cooking phases, or as complex as code to control a family of sensors and actuators at a low level. Portability arises as a significant issue in these systems; for instance Java got its start as a language for portable programming of set-top boxes. Extensibility concerns the ability of the programming system to accommodate changes in the underlying device. For instance, the 10 digit recipes do not take account of whether the microwave has a turntable; indeed, one wonders if step 4 in the recipe in Figure 4 makes sense in this case. Ideally, programming APIs for embedded systems will be like those for PCs and assume that new peripherals will be introduced from time to time and access will be provided through some API. A rigid system may need to be completely redesigned to take advantage of a new sensor or actuator. Predictability is similar to the challenges with security and performance for programmable network elements. However, the concerns are often different. For example, a microwave may have little security risk but significant safety risk. Wireline routers are very performance sensitive, but typically do not care about power utilization in the way a cell phone might. Microwave ovens are not sensitive to power or performance, but typically are very sensitive to cost (in dollars) and convenience for unskilled operators. An interesting issue is whether domain-specific assumptions provide a new handle on predictability.

For instance, it is undecidable whether a program in a GPL will cause a variable to exceed a given value, but it is trivial to tell how much cook time the recipe in Figure 4 will require. Finally, as argued earlier, the complex connectivity of embedded system raises deliverability mechanisms as an interesting question.

3.2. Mechanisms

Java provides a good example of several of the mechanisms that can address requirements for mobile code. A core question is whether a DSL is needed or whether an existing language can be used if it is provided with a suitable development environment or runtime analysis system. In the case of Java it was decided to produce a new general-purpose language that prioritized portability and security over performance and compatibility with existing C libraries. Subsequent efforts to use Java in diverse contexts have focused on using sandboxing and the JVM or something near to it, like the KVM, to address the needs of specialized contexts, like web browsers, mobile communication devices, or embedded systems. Much of the work on active networks has focused on the use of Java, particularly in the ANTS EE. ANTS provides a limited library API for Java and support for its code distribution system based on identifiers and on-demand installation. Many issues with security are addressed by the limitations imposed by sandboxing and limited APIs. For instance, a packet that wishes to be processed by a given program uses the cryptographic hash of that program to identify it; this prevents any confusion that might arise if an installation request attempted to spoof an often-requested network service. This approach exacts a price in configuration management, however, when a common service needs to be transparently upgraded: existing programs at endpoints will need to be modified to use the hash of the new program. The use of an existing scripting language or GPL can exploit existing support for development and runtime systems. A special-purpose approach like SmartPackets must build a new compiler and/or runtime system for its language, whereas ANTS can use off-the-shelf compilers and the JVM. A hybrid approach like SwitchWare requires an interpreter for PLAN, but can make use of the OCaml runtime system for its services. The hybrid strategy has the advantage of allowing functions to be 'pushed down' from the invocation layer to the service layer if they are more appropriate for implementation in a GPL.

The advantage of a DSL is the simplicity of the programs and the ability to exploit this simplicity to achieve other objectives like demonstrating security properties. Moreover, DSLs enable significant kinds of innovation in the way the system works without the baggage associated with a GPL. PLAN provides numerous examples of this, as we discuss further in Section 4. One specific area of challenge is in *resource control*, that is, the ability to determine and limit the use of a valuable resource by a mobile program. Java provides limited resource control by preventing programs like applets from carrying out potentially troublesome operations like accessing files on the disk of the browser. However, Java does little to prevent the use of space or cycles on the host machine. It is important to break this problem down into two architectural options to clarify

the tradeoffs: *usage limitation* or *bound verification*. Under usage limitation the mobile program is given a collection of resources such as a given amount of space or cycles; if the program exceeds these limits it may be terminated or otherwise limited, for example, by having its priority reduced in cycle scheduling. In bound verification the program is checked in advance to determine whether it satisfies the necessary limitations. If it does, then it can be run with more limited runtime monitoring. Bound verification has significant advantages over usage limitation not only because it may impose less burden on the runtime system but also because usage limitation essentially begs the question of how it is known that a program will meet its usage limits and therefore perform properly. However, bound verification must be based on a technology for verifying the desired property. Static type checking is a major success of this approach, but verifying space and cycle usage are more stubborn problems. An additional problem that arises in programmable networks is the need to deal with decentralized replication of mobile programs that run on multiple routers. In this case usage limitation is problematic since no system has global knowledge of usage. This problem was recognized early in the TCP/IP system and addressed with the TTL value, which keeps track of how many more routers the packet should be allowed to visit.

In realizing bound verification there are essentially two options; these can be broadly classified as verification 'by others' versus verification 'by me' (that is, my local trusted computing base). The former is seen in systems like Microsoft Authenticode, which attaches a digital signature to code as proof of its conformance to requirements. That is, the code consumer trusts the code because it came from a trusted source. ActiveX controls aim for this kind of verification. By contrast Java applets aim more toward verification on the code consumer machine, eliminating the need for a trusted origin (at non-trivial cost to functionality). Efforts have been made to extend this approach by the use of proof-carrying code [13, 12], in which evidence of conformance is included with the mobile program. This evidence can be used to verify conformance by the code consumer.

Micro mobile programs provide another mechanism for addressing requirements. For delivery they offer the option of providing complex information through 'in-band' delivery. For instance, a small special-purpose diagnostic program that can be written in a few dozen lines can be sent in a packet to perform its task. In a system like ANTS, an 'out-of-band' delivery mechanism would install the program on each node and then send a new packet to invoke it by hash index. In a system like a microwave that reads a UPC barcode and looks up a program to match, the device requires connectivity to the Internet in order to maintain its selection of programs; a micro mobile program in a 2D barcode can provide all of the needed code without this connectivity. For predictability, micro mobile programs offer the prospect of carving out a class of programs for which analysis is feasible. This is the case for both micro DSL and GPL programs. The next two sections illustrate some of the ideas in each of these cases.

4. Active Packets

PLAN is a small scripting language with a syntax and semantics similar to Scheme and ML. Implementations have been carried out in several languages, but the reference implementation is written in OCaml and assumes a service layer of OCaml programs. The essential design goal was to balance ephemeral code in packets with resident service-layer code. Thus the nature of programming with PLAN is to decide what goes in the packet in PLAN versus what gets written in OCaml and installed on a node. For instance, a simple diagnostic or configuration packet that is meant to be executed once on each of a family of active nodes is written in PLAN, whereas a program that is complex, requires significant state or timers, or needs to be used many times is best coded in OCaml, installed as a library service on nodes and invoked from PLAN. PLAN programs therefore focus on simple invocations of service layer programs or provide discovery and set-up functions.

Perhaps the purest illustration of PLAN, and one of the purest illustrations of active networking generally, is the *PLANet* network testbed [7]. PLANet implements a range of internetworking functions, including both standard IP functions like distance vector routing and novel functions like Flow-Based Adaptive Routing (FBAR). FBAR allows PLAN agents to discover QoS properties and configure customized routes. The implementation of PLANet is in OCaml and essentially replaces the usual network layer with an active network functionality. Thus all packets are PLAN programs wrapped within link layer frames. The character of programs in PLANet reflects the tradeoffs between deployment of programs in the invocation versus service layers. For example, FBAR functions that perform diagnostic searches for good paths and set up labeled routing are written in PLAN, whereas distance vector routing, which involves state with tables and times, is written mainly in OCaml but uses PLAN functions to send routing table advertisements.

Another principal rationale for PLAN was the hope that a DSL would provide better support not only for convenient coding but also for reasoning about properties of programs. PLAN programs are comparatively simple and can be constrained to display desirable properties so specifying their semantics and reasoning about them is easier than doing so for OCaml or any similar GPL. Accomplishments along these lines include the specification of PLAN using a term rewriting model [9] and formal reasoning about FBAR [18]. Aside from this, there have been two main lines of investigation on reasoning about PLAN: resource control and the impact of programmability on communication privacy.

Resource control is a significant problem for programmable networks. The IP protocol provides for a TTL field to prevent packets from cycling indefinitely in the network. PLAN uses a similar concept and adds to this a guarantee that PLAN programs terminate if the services they invoke terminate. This is done simply by not including recursion or looping constructs in PLAN, a reasonable tradeoff given that network programs often do not need looping constructs and, when they do, these can be included in service layer functions. ANTS also provides for a TTL-like resource bound, but treats it more liberally than PLAN to

support functions like multicast. This enables an exponential blow-up in program proliferation that would probably be as bad in practice as a completely unbounded program. As for PLAN, the resource bound on packet proliferation is more strict, but an individual packet with a nested collection of function definitions can display exponential use of time and space on a node [9]. A more recent direction is to use a special-purpose byte code called SNAP [10], which can be compiled from PLAN [8] or another source. SNAP shares with PLAN the property that resource bounds can be predicted from program lengths (so resource utilization is governed by bandwidth) but SNAP enables tighter estimation of the bounds [11].

Another line of study concerns the impact of active network functionality on guarantees of privacy. A PLAN packet can enter a network, gather diagnostic information, leave configuration state (if service layer functions support this), and return to its origin without needing to visit anything but active routers. This is clearly somewhat different from IP packets, which can invoke ICMP responses but otherwise have little ability otherwise to collect and return information from routers. There is work [6] exploring how to reason about the ability to collect information from active networks for various assumptions about available service layers, including reasoning about strategies for corrupting routing functions using active packets. More recent work investigates topics like anonymity and onion routing in active networks.

5. Active Barcodes

Barcodes provide an extremely cheap way to communicate bits. They can be printed on paper and therefore do not require any special material to be produced; a small, robust reader can be had for a modest cost. Linear barcodes are used very commonly for postal addressing, inventory management, and point-of-sale functions. 2D barcodes are a newer technology that has been making headway in various applications such as postage, where information in the barcode can be used to provide evidence of payment. The US Postal Service, for example, has explored the idea of digital signatures in 2D barcodes as part of its Information-Based Indicia Program (IBIP). Putting programs into barcodes is an idea explored in [4]. We discuss here some of the issues that arose in that study and how they compare to other applications for micro mobile programs.

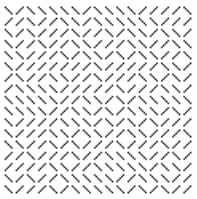

Figure 5. Enchilada Recipe as a Dataglyph Symbol

The nature of active barcodes is likely to depend heavily on the application domain. In the case of microwave programs it is possible to put a program like the enchilada program of Figure 4 into a 2D barcode. Figure 5 shows the result of doing this, where the program has been coded in Java, reduced to bytecode, and compressed. The program in the barcode here is actually somewhat more sophisticated than the one in Figure 4. It includes a feature that modifies cooking times based on cooling that occurs after the user pauses

the microwave: in particular, it keeps track of how long it takes the user to perform steps 3 and 4 and adjusts the cook time in 5 accordingly. It would be inconvenient for the user to do this himself. If the oven has a rotating platform then the program causes it to rotate the food and omits step 4, that is, does not ask the user to do the rotation. Thus the program exploits in interesting ways the flexibility offered by the opportunity to deliver a micro mobile program.

There are at least two interesting research problems that are suggested by this application. First, what can be done to compensate for the changed operator involvement? For example, if the recipe is miscoded to indicate cooking for 145 minutes rather than 1 minute and 45 seconds, a user is likely to notice this, but this sanity check may be missing if the program is not read by the operator. Second, how much of what kind of code can or needs to be used in the barcode? For example, it seems impractical to locate a Java JIT on the microwave, and it is not clear whether compression, for example, will be of any value for such small programs.

Interaction with human operators and the physical environment are two recurrent themes for embedded systems. The nature of these interactions is likely to be somewhat domain specific. In the microwave example, almost all of the non-determinism in the program is created by user actions. This particularly contrasts with assumptions in many other applications, where non-determinism arises from concurrency, and this has ramifications for predictability. As mentioned before, it is straight-forward to calculate maximum and minimum cooking times from the program in Figure 4. Doing this for the one in Figure 5 is harder, but not nearly as hard as a comparable task might seem for an arbitrary Java program. The problem is similar to array bounds checking: one needs to verify that a value is never more than a certain value in any run of the program. In this specific case this, depends on the operator interaction because the feature of the program that adds back cooking times would prevent the program from ever completing if the operator continued pausing it indefinitely. However, it is feasible to apply reasonable operator assumptions and prove, with off-the-shelf formal analysis tools, that the program does not cook the food for any more than a certain length or *any less* than a certain time (this is also a safety issue if the food is raw). These formal analysis techniques can be somewhat automated so it is possible to create an architecture in which the burden of verification is placed on the (sophisticated) development environment, and the code consumer can use mainly usage limitation to predict behavior. Thus the basic program is converted into one that keeps a cook time counter and rejects program runs that use less or more than a pre-specified range of times.

The question of what kind of code to put in the barcodes is interesting, but again domain-specific. For instance, even these simple recipe programs can be more clearly written in a reactive programming language like Esterel than in Java. However, portability is likely to be a key consideration in these applications (recall the leftist perspective in Figure 2) so the use of a highly portable GPL has advantages. This suggests that shipping JVM bytecode is a plausible approach, especially if the development and analysis environment

and perhaps the target device can take advantage of the fact that the micro mobile programs can probably use only a modest fragment of Java. Shipping portable byte code rules out a number of options for how to compress the code since schemes based on source code or modified versions of the JVM can be ruled out. It is not obvious that a Java bytecode of only about a 1000 bytes will compress at all well given the overhead associated with compression. The situation is similar to that for IPSec-level compression [16], where each packet (of about 1300 bytes) must be individually compressed. Fortunately Java byte code recipes seem to display significant redundancy: for the example of Figure 5, a compression technique called Pack [14], specifically designed for Java bytecodes, compresses the 894 byte Java enchilada program to 60% of its original size.

6. Conclusions

Micro mobile programs are useful and feasible in a variety of contexts. Moreover, there are a number of recurrent themes that enable ideas in one context to be inspirational in others, even when there are significant differences in the kind of application involved. One central themes is the challenge of flexible open APIs that have predictable behavior. Micro mobile programs can help attain this objective, especially if domain-specific circumstances can be identified that aid the analysis of programs when they are being developed or at the time that a DSL for the micro mobile programs is designed.

Acknowledgments

The author is grateful for the insights and experimentation of participants in the SwitchWare and MiRL projects. The work was partially supported by DARPA (N66001-96-C-852), ONR (N00014-99-1-0403 and N00014-00-1-0641), and ARO (DAAG-98-1-0466 and DAAD-19-01-1-0473).

References

[1] Adobe. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[2] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The switchware active network architecture. *IEEE Network Magazine*, 12(3):29–36, May/June 1998. Special issue on Active and Controllable Networks.

[3] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A secure active network architecture: Realization in SwitchWare. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, 1998.

[4] Alwyn Goodloe, Michael McDougall, Rajeev Alur, and Carl A. Gunter. Predictable programs in barcodes. `http://www.cis.upenn.edu/sdrl/mirl/papers/predictable_barcodes.ps`, April 2002.

[5] Carl A. Gunter, Rajeev Alur, Alwyn Goodloe, and Michael McDougall. Third-party programmability for embedded processors, December 2001.

[6] Carl A. Gunter, Pankaj Kakkar, and Martín Abadi. Reasoning about secrecy for active networks. In Paul Syverson, editor, *13th IEEE Computer Security Foundations Workshop*, pages 118–131, Cambridge, England, July 2000. IEEE Computer Society.

[7] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society Infocom Conference*, pages 1124–1133, Boston, Massachusetts, March 1999. IEEE Communication Society Press.

[8] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Compiling PLAN to SNAP. In *Proceedings of the IFIP-TC6 Third International Working Conference, IWAN 2001*, September/October 2001.

[9] Pankaj Kakkar, Michael Hicks, Jonathan T. Moore, and Carl A. Gunter. Specifying the PLAN networking programming language. In *Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999. `http://www.elsevier.nl/locate/entcs/volume26.html`.

[10] Jonathan T. Moore. Safe and efficient active packets. Technical Report MS-CIS-99-24, Department of Computer and Information Science, University of Pennsylvania, October 1999.

[11] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, April 2001.

[12] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[13] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[14] William Pugh. Compressing java clas files. In *ACM Sigplan Conference on Programming Language Design and Implementation*, pages 247–258. ACM Press, 1999.

[15] Beverly Schwartz, Wenyi Zhou, Alden W. Jackson, W. Timothy Strayer, Dennis Rockwell, , and Craig Partridge. Smart packets for active networks. In *Proceedings of the Second IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, pages 90–97, March 1999.

[16] A. Shacham, R. Monsour, R. Pereira, and M. Thomas. IP payload compression protocol (IPComp). RFC 2923, IETF, December 1998.

[17] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[18] Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal verification of a PLAN algorithm in Maude. In Tenh Lai, editor, *Proceedings of the 2000 ICDCS Workshop on Distributed System Validation and Verification*, pages E:49–E:56. IEEE Computer Society, April 2000.

[19] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the First IEEE Conference on Open Architectures for Signalling (OPENARCH)*, pages 117–129, April 1998.