A Fresh Look At Low-Power Mobile Computing

Michael Franz
Department of Information and Computer Science
University of California, Irvine

Abstract *We challenge the apparent consensus that next-generation mobile devices must necessarily provide resource-intensive capabilities such as Java interpreters to support advanced applications. Instead, we propose an architecture that exploits the high "last mile" bandwidth in third generation wireless networks to enable the largest part of such applications to run inside a base station, effectively reducing the mobile device to a dumb terminal. We discuss some implications of this architecture, with respect to hand-off, confidentiality while roaming in potentially hostile networks, and the need for a server-transparent segmentation of applications into a computational and a user interface component.*

1. Introduction

Latency and bandwidth continue to lag behind demand, and this situation will continue even with the availability of third generation mobile networks. In future wireless networks, available bandwidth between the mobile terminal and the base station increases significantly, but this doesn't solve the problems of bandwidth contention further upstream or those relating to latency, especially when dealing with remote applications.

The traditional solution of overcoming these problems has been to place more local intelligence at the user's site. The most successful approach to this so far has been that of downloadable "applets" or "mobile agents" that travel from a server to the user's terminal device and conduct most of their computations locally, unhindered by bandwidth or latency constraints.

Unfortunately, downloading and executing mobile code requires considerable resources, both computationally as well as storage wise, with an immediate implication on power consumption. At first sight, this might rule out mobile-code client solutions targeted at low-cost low-power handheld devices such as next generation mobile voice/data terminals and networked embedded processors in consumer products.

We propose a solution based on off-loading the resource-intensive parts of the client program to the mobile base station[1] or to an appropriate computational device in the user's personal sphere, such as a "home base station" or a more capable notebook/PDA computer carried on the user's body and connected via a technology such as Bluetooth. In the most extreme case, the user's mobile device then only needs the functionality of a dumb terminal, making use of a high available bandwidth between the actual execution unit (XU) and the mobile station (MS)—but the MS might also perform critical functions locally, perhaps utilizing reprogrammable hardware. The original two-level client-server architecture thereby becomes a three-level MS-XU-server architecture.

A key characteristic of the proposed architecture is that a remote server need not be aware of the fact that the client is not actually running on the mobile station. Further, the division of concerns between MS and XU can be adjusted, potentially even at run-time, without notifying the server of this fact. Finally, local state is kept at the XU (albeit possibly partially cached at the MS), increasing overall robustness in the event of power failure of the MS or intermittent communication breakdown.

The following sections describe the proposed architecture in more detail and point to areas that require more research. Among other facets, we discuss the implications of handover and the resulting need to transfer state between different XUs. Of particular interest to service providers might be that the provision of computational resources at the XU constitutes a possible source of revenue, and that control over such user state might tie users closer to providers and thus reduce churn. We discuss the issue of automated segmentation of functionality, by

[1] In a GSM system, the entity performing these resource-intensive tasks would most probably be co-located with the MSC (or rather, be part of the Serving GPRS Support Node) while in cdma2000, these functions would fit within the BSC node.

which a controller decides which parts of the client program should be downloaded to the MS and which parts should run at the XU. In case of a MS that includes reconfigurable FPGA hardware, historical usage profiles might be used to select an initial configuration of the FPGA grid, but this could be updated in real time (using dynamic recompilation) under the remote control of an XU that constantly models the execution of the reconfigurable hardware in the MS. Lastly, we discuss the fact that certain security applications such as banking might require dedicated functionality (such as encryption) to be downloaded to the reconfigurable hardware in order to guarantee end-to-end functionality, and how this might be modeled in our architecture.

2. Architecture

A common approach to masking the latency and limited bandwidth of client-server applications has been the addition of local processing power at the client's site ("fatter" clients). The spectacularly successful Java platform is the current champion of this paradigm. Java's inventor, Sun Microsystems Inc., has been trying to extend the reach of the Java platform ever further down the performance spectrum to embedded consumer devices such as PDAs, smart-phones, or even household appliances, by introducing a core (*picoJava*) for dedicated processors that can execute Java bytecode directly on the silicon.

Processors based on the picoJava core require far less external memory than alternate solutions based on general-purpose processors for interpretation (or even dynamic translation) of Java bytecode. Unfortunately, however, picoJava's implementations are likely to be quite complex and consequently relatively expensive and power-hungry. Hence, cost and battery-life considerations reduce the appeal of increasingly "smart" mobile terminals.

We propose an alternative architecture (Figure 1) in which the client program is further subdivided between an execution unit (XU) and a mobile station (MS). For example, and this is our main target scenario, the XU could be incorporated into a mobile service provider's infrastructure and the MS could be a mobile terminal—but the XU might also be a television set-top box (or network-connected games console) and the MS a remote control unit. The key characteristic of our model is that there is a direct (i.e., very low latency) connection between XU and MS, and that this connection has sufficient bandwidth. Given these parameters, the MS in the most extreme case could then act simply as a "dumb terminal".

In our model, the server is unaware of the fact that the client is actually composed of two distinct parts, XU and MS. To the server, the XU-MS pair acts as an atomic entity that is physically located wherever the MS happens to be (for location-aware services) and that has at least the computational resources of the XU (the MS may contribute to the total). Hence, in the Java case, a JVM class file would appear to be downloaded to the MS, but most of it would actually be executed on the XU, with the segmentation between the two either being hard-coded into the libraries on the XU (simple case) or dynamically decided and adjusted using just-in-time code generation (a much more challenging case).

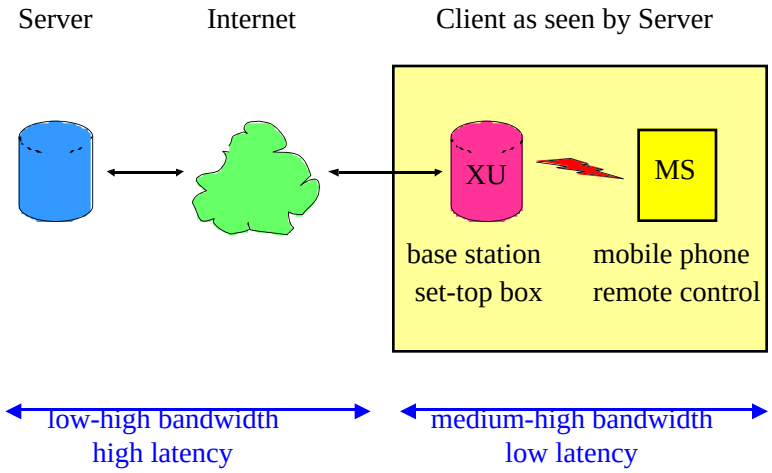

Figure 1: System Architecture

3. Handover and the Quantization of Computational Resources

A cornerstone of our architecture is the provision of a low-latency connection between the XU and the MS. In the case of a MS that is physically moving, this implies moving the state of the computation that is running on the XU along with the MS to keep it "close" and the latency low. While one might also envisage a solution in which the computation remains stationary on the initial XU and all traffic is routed via that XU, perhaps using a fixed-bandwidth reservation scheme, this would clearly be sub-optimal; only true mobility of the computations themselves will result in fully scalable performance.

Hence, it becomes necessary to transfer state from one XU to another. This brings with it the potential to have XUs in different organizational domains seamlessly working together: For example, while the user is physically at a home or office location, execution services might be provided more cheaply by a private server installed on those premises. As a user enters a locality that offers such a (cryptographically authenticated) execution environment, the whole active state is transferred away from the service provider and is eventually restored there as the user leaves the locality; all of this completely transparently to the user. Figure 2 shows some of the possible scenarios. Note that in such a transfer from service provider to local context and back, the service provider can actually cache the local state, so that only the changes need to be communicated as the MS re-enters the domain of the service provider.

A third scenario depicted in Figure 2 is that of a user carrying his or her own mobile context that is sufficiently capable computationally to carry out the functions required of an XU. This is not really all that far-fetched. For example, the user may be carrying a PDA or laptop in close proximity that is sitting idle while the user is making a flight reservation using his or her mobile phone. It might make sense off-loading the computation to the local device while the user is, for example, sitting on a train for several hours. The problem here is that the device hosting the local context may itself run out of battery power or else may be shut off at any time; this suggests that the signaling of the upload/download actions would most probably have to be done by the end-users themselves and that fault-tolerance considerations need to be incorporated into the final design.

This brings us to the issue of the actual transfer of the local state, which requires all of the following:

1. The receiving XU must have sufficient resources to be able to add this computation to its existing workload.
2. The receiving XU must have the required libraries available locally to support the ongoing computation. Note that an XU that is underused can prefetch libraries used at "neighboring" MSCs in anticipation of future handovers.
3. There must be sufficient network resources to enable the transfer of state information between

Execution Unit	Scenario
network/provider-based	Network service provider additionally also supplies computational infrastructure, probably for a fee. Handover requires continuous transfer of computational state "staying close to the mobile station" in order to preserve latency characteristics. Unlike voice calls, which are dropped when the current provider's network is left, it would be desirable to provide continuous handover of computational state even when roaming across different service providers' networks.
user's mobile context	User carries an additional device such as a laptop or PDA that can provide computational services to the MS and that is linked to the MS using some body-area networking technology such as Bluetooth. Upload/Download of state both from/to external service provider and/or home base station possible under user control; similar to "synching" today's PDAs.
home/office base station	As user enters home or office, active computational state is transferred from network provider's computers to a local execution unit installed on private premises, relieving the network provider of the obligation to host the computation and temporarily suspending billing for these services.

Figure 2: Different Classes of Execution Units and Applicable Usage Scenarios

the two XUs within a useful time frame. But note that the timing constraints for this sort of handover are much less stringent than they are for voice connections, and that they need not even occur simultaneously with an ongoing voice handover. In fact, if one allows for a slight temporary increase of latency, one may permit a nomadic user to move a certain number of "hops" away from its current XU and hand over the computation only when the number of hops exceeds a certain threshold or the user becomes stationary. This approach may be particularly attractive when complemented by the previously mentioned idea of a "user-carried" XU that might be used, e.g., on a longer high-speed train journey.

3.1 Standardization of Execution Environment's Parameters

We believe that a general adoption of our architecture would most probably lead to a standardization of the execution environment's parameters, not just in terms of the libraries available (as in Sun's JavaOS project) but in terms of quantifying actual processing and memory resources in a standardized way independent of an actual processor platform. Such standardization not only aids in allocating resources when transferring computations between XUs, but might also be useful for billing purposes. For example, a user might subscribe to a guaranteed minimum workspace of 25kObjects and a minimum throughput of 50kTicks per minute, with specified penalties for going over those limits. A combination of paid-for-resource allocation and dynamic resource control is also an effective defense against denial of service attacks.

In our initial approach to exploring these issues, we are using the Pi calculus and the *Pict* programming language [PT97] for systematically modeling the connectivity, communication and mobility (MS moving from one XU to another) arising in our architecture.

3.2 A Commercial Vision: Impact on Billing, Customer Loyalty and Churn

Most likely, the provision of computational services could be turned into a significant source of revenue for service providers. On one hand, off-loading the computations into the network makes for extremely cheap mobile devices that can be distributed without a subsidy—"disposable" devices are entirely feasible. On the other hand, it would enable a greater range of services to be provided in the first place that might otherwise bypass mobile service providers completely. Furthermore, the available level of network-based computation might turn into a key service distinction between competing providers.

Of note is also that many of the envisioned *applications* are of the "always on" kind, i.e. the application programs never "quit". These could be structured in such a way that they increase customer "stickiness". For example, the user might have two applications implementing an electronic organizer and an MP3 jukebox, both of which are permanently running on his MS, hosted by the actual service provider. Although these are in reality downloadable applications like any other, they could be marketed as a basic service (i.e., not count towards the allocated "clicks" and "object space") and as far as the user is concerned might just as well be built into the MS itself. As a consequence of this marketing strategy, the user will not choose a competing service from an external source. But now, all user-defined preferences and settings are "owned" by the network service provider, reducing the users' propensity of changing providers.

4. Segmentation of Functionality: The XU-MS Split

Client applications are jointly executed by the XU and the MS, but creating the illusion to the server that all of the processing happens at the geographic location of the MS. This leads to the question how functionality is divided between the XU and the MS.

In the most simple case, the MS would simply be a dumb terminal. The XU would run a "virtual terminal" that translates GUI commands coming from the client application to remote-terminal instructions for the MS. Since all interaction with physical I/O devices in Java is via well-established libraries, this would be very simple to accomplish in the Java case. We also note that others have been successful at building an architecture in which MS Windows GUI commands are intercepted on a server and relayed to a client residing within a web browser [Citrix], enabling the remote execution of local desktop software from anywhere on the web—this is essentially the exact same architecture as our proposed "dumb terminal" solution.

A much more challenging objective would be to actually perform part of the processing on the MS itself. For the purpose of minimizing power consumption on the MS, one needs to include the

power used for communication in the overall cost equation. Using a "dumb terminal" approach, the mobile station is likely to be receiving much more data than it needs to send, but there may still be significant number of handshakes and other data exchanges that might be reduced using local processing on the MS. Note that the XU will be able to model current and future power consumption of the MS quite accurately as it gathers usage statistics on a per-application or even per-user basis.

The task of partitioning the application between a resource-constrained MS and a remote XU is not unlike the task of partitioning an embedded system (between slow but flexible software, and fast but rigid hardware). There are also some similarities with partitioning for parallel execution, particularly considering the fact that some partitionings may require more communication among the participants than others. Hence, it is quite likely that ideas from the embedded systems community and the parallel/distributed systems community can be adapted to this problem domain.

It is also important to note that the current version of JVM-code is unlikely to survive as the dominant mobile-code interchange format in the longer-range future. As we have demonstrated in previous research [FK97, ADF01, ADR01], there are alternative formats not based on virtual machines that provide much better performance than JVM does, and more easily verifiable security. It is likely that such future formats would include annotations that would guide in the partitioning of functionality between XU and MS. We are currently working on mechanisms for integrating specific annotations describing programmer-specified parallelism and compiler-derived parallelism into a mobile code format in a manner that is useful to a parallelizing compiler back-end at the code recipient's site.

4.1 Use of Field-Programmable Hardware in the Mobile Station

The deployment of field-programmable hardware in mobile stations then becomes an interesting possibility. Ideally, the same analysis process that partitions functionality between the XU and the MS could be taken one step further, creating custom hardware on-the-fly on the MS where this is most beneficial. Such downloaded dedicated routines might include encryption protocols, vocoders for speech (input, output, and telephony), and visual-object rendering (texture mapping etc.) capabilities for gaming applications.

The availability of an appropriate (non-virtual-machine) intermediate representation makes this task considerably simpler. For example, at UCI we have been experimenting with mobile-code representations based on the *condensed graph model* [Mor96]. Condensed graphs express Availability-Driven, Coercion-Driven and Control-Driven Computing in a single unified model of computation and have transformations that allow you to "turn up or down" the amount of processing power required. Essentially, this transforms the partitioning problem from a speculative, data driven model using a substantial amount resources to a lazy, demand driven model that tries to minimize resource usage by applying simple topological transformations.

The availability of such a model in the mobile-code representation itself significantly reduces the burden on the XU (or appropriate controller) for computing such a model, and makes it possible to even adjust these bindings on-the-fly in reaction to changing user activities on the MS: based on a model of what the MS is doing, the XU at regular intervals sends it an updated FPGA configuration.

4.2 Special End-To-End Application Requirements

Lastly, we observe that there are applications that require special care in partitioning. For example, communications with a banking application are encrypted, with the assumption being that (1) the client program is actually (atomically) executed at a single terminal location and (2) that this terminal location provides a trusted computing infrastructure, i.e., the data path between the mobile client program and the user (through the OS, web browser, etc.) is not corrupted by an interposed malicious agent.

Obviously, condition (1) is violated in our architecture, and indeed the data path between the XU and the MS is potentially much longer and subject to eavesdropping and corruption. Worse still, when roaming in foreign networks, any potential secret would be readily available to the foreign entities running such networks simply by observing the XU.

A solution to this problem lies in designing an appropriate type system into the mobile-code transportation scheme itself that can directly express a notion of confidentiality and trust. The XU is then under the obligation to partition the mobile client program in such a manner that the trusted parts reside on the MS. This property can be verified using proof-carrying authentication protocols [AF99] incorporating the code being downloaded itself

—after download, the MS generates a response sent back to the server that vouches for the fact that the trusted subset of the computation is now running on a trusted host.

5. Status and Research Vision

We are currently embarked on a substantial project (funded by NSF and DARPA) to design a new mobile-code distribution architecture that reconciles provable security with execution efficiency.

A first major result of this project is our discovery of a class of mobile code representations that can encode only programs that provably cannot harm the target machine—these mobile code formats therefore provide security "by construction" rather than "by verification" as currently required by Java [ADF01, ADR01].

Our project also addresses some of the issues of functionality segmentation, as well as the required protocols to perform mobile-code verification, dynamic translation, and execution at perhaps physically disjoint sites. However, it does not currently address the handover issues we have brought up here, nor does it currently address field programmable hardware.

Our vision is to elevate good power management into a system-wide quality of mobile-code systems. We believe to have identified several topics requiring further research and are already embarked on the quest for solutions.

Acknowledgement

This paper was inspired by an afternoon that the author spent at the TIK institute of ETH Zurich talking to B. Plattner, L. Thiele, and their associates. The author would also like to thank the anonymous referees for their constructive comments.

This research effort is partially funded by the U.S. Department of Defense, *Critical Infrastructure Protection and High Confidence, Adaptable Software (CIP/SW) Research Program of the University Research Initiative* administered by the Office of Naval Research under agreement N00014-01-1-0854, and by the National Science Foundation, *Program in Operating Systems and Compilers*, under grant CCR-9901689.

References

[ADF01] W. Amme, N. Dalton, P. H. Fröhlich, V. Haldar, P. S. Housel, J. v. Ronne, Ch. H. Stork, S. Zhenochin, and M. Franz; "Project transPROse: Reconciling Mobile-Code Security With Execution Efficiency"; in *The Second DARPA Information Survivability Conference and Exhibition (DISCEX II)*, Anaheim, California; IEEE Computer Society Press, ISBN 0-7695-1212-7, pp. II.196–II.210; June 2001.

[ADR01] W. Amme, N. Dalton, J. v. Ronne, and M. Franz; "SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form"; in *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI 2001)*, Snowbird, Utah, pp. 137–147; June 2001.

[AF99] A. W. Appel and E. W. Felten; "Proof-Carrying Authentication"; in *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore; November 1999.

[Citrix] Citrix, Inc.; *Citrix® Independent Computing Architecture and Citrix MetaFrame™*; http://www.citrix.com.

[FK97] M. Franz and T. Kistler; "Slim Binaries"; *Communications of the ACM*, 40:12, pp. 87–94; December 1997.

[Mor96] J. Morrison; *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*; Technische Universiteit Eindhoven, ISBN 90-386-0478-5; October 1996.

[PT97] B. C. Pierce and D. N. Turner; "Pict: A Programming Language Based on the Pi-Calculus", in G. Plotkin, C. Stirling, and M. Tofte (Eds.), *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press; 2000.