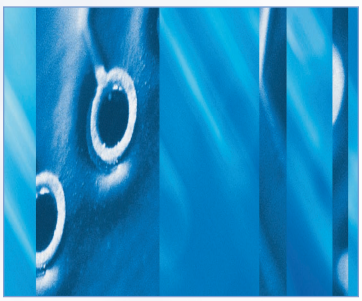
Challenges and Lessons in Developing Middleware on Smart Phones

Several research projects pursuing middleware architectures to support pervasive applications on smart phones reveal the importance of careful resource management, lightweight communication protocols, and asynchronous programming.



Oriana Riva
ETH Zürich

Jaakko Kangasharju
Helsinki University of Technology

Smart phones are the most promising heralds of future pervasive computing. With increasingly powerful capabilities in computation, communication, and sensing, these devices have evolved from simple communication devices into consumers of various types of information services as well as sensor hubs for healthcare monitoring systems, sports trainers, and the like.[1]

Programming phone applications that can properly perform in this new computing landscape is not easy. Middleware platforms that can, for example, abstract the complexity of network communication, fault tolerance, and component migration are useful tools to aid the development of distributed applications. However, while middleware traditionally seeks to provide a useful layer of abstraction, an important design principle on phones must also be resource awareness, especially of resources such as energy that are not a primary concern in desktop computing. Middleware for smart phones calls for novel approaches.

During the past five years, we have participated in several research projects at the Helsinki Institute for Information Technology (www.hiit.fi) that focus on middleware for pervasive applications. We have explored various topics ranging from XML messaging and synchronization, to event-based communication and service migration, to context monitoring and reconfiguration. We have also built several prototype systems running on modern phone platforms, and we have evaluated their performance in experimental testbeds, in some cases organizing field trials for more extensive evaluations. Based on these experiences, we have identified several middleware research challenges along with possible solutions.

SMART-PHONE CHARACTERISTICS

Smart phones are relatively powerful mobile computing platforms. They offer reasonable data-processing and -storage capabilities, incorporate various communication technologies, and often include embedded devices such as cameras or sensors. The most useful feature of smart phones as enablers

0018-9162/08/$25.00 © 2008 IEEE

of pervasive computing is the possibility of installing new applications, which in many cases anyone, not just the manufacturer or operator, can write.

Hardware

Smart-phone CPUs generally use the ARM architecture due to its power efficiency. Clock frequency in modern models is usually around 200-250 MHz, but recent high-end models exceed 300 MHz. Total RAM is normally between 64 and 128 Mbytes, but programs can use only a fraction of this, typically 10-20 Mbytes. External storage is provided by flash memory cards with capacity up to 8 Gbytes, but because smart phones' operating systems do not support virtual memory, this cannot automatically be used as an extension to RAM.

Most smart phones use lithium-ion batteries due to their small form factor. Charge ranges between 780 and 1,200 milliamp hours (mAh), which is sufficient for several days of standby operation or a few hours of phone calls.

The primary data communication technology on smart phones is the mobile phone network operated in packet-switched mode, usually either Enhanced Data Rates for GSM Evolution (EDGE) or the Universal Mobile Telecommunications System (UMTS), although some networks only have the lower-speed General Packet Radio Service (GPRS) available. Bluetooth and infrared technologies are available for short-range communication, but infrared is rarely useful due to its line-of-sight requirement. Modern business models increasingly also offer Wi-Fi (IEEE 802.11b/g) connectivity.

The most useful feature of smart phones as enablers of pervasive computing is the possibility of installing new applications, which in many cases anyone can write.

A modern smart phone is not only a communication device but also functions as a media center with the inclusion of cameras as well as music and video players. Global Positioning System (GPS) receivers are available as external accessories and are integrated into some recent phone models. Screen size and resolution, keyboard size, and keypad design have also advanced, facilitating applications such as Web browsers, e-mail, and video players.

Software

The most widely used operating system on smart phones is Symbian OS, with more than 70 percent of the market share. Symbian is an open platform specifically designed for resource-constrained mobile devices, with a focus on small memory footprint and low energy consumption. Symbian's native programming language is a dialect of C++, but it supports Java and other programming languages such as Python, Visual Basic, and Perl as well.

Smart phones use the Java Platform, Micro Edition (http://java.sun.com/javame/technology), a reduced version of the Java Platform, Standard Edition (Java SE), that is designed to cope with small devices' resource constraints. Conceptually, Java ME is divided into *configurations*, which provide the requirements on the device hardware and Java virtual machine (JVM), and *profiles*, which provide the APIs and libraries available to the application programmer. On phones, two standard configurations are available.

The Connected Limited Device Configuration is designed for smaller devices. CLDC is a severely restricted version of Java, with older versions not providing support even for floating-point types. It works in conjunction with the Mobile Information Device Profile, a stripped-down version of the Java SE 1.1 library with MIDP-specific user interface and networking libraries. To save resources and still retain Java's safety properties, the compiler performs most of the required class file verification during the preverification phase.

The Connected Device Configuration is designed for more-capable devices. CDC profiles are arranged in a stack, with the Foundation Profile providing an almost complete Java SE 1.1 environment enhanced with the Collections API. On top of this, the Personal Basis Profile and Personal Profile provide UI libraries.

Standard practice for software development on smart phones is to write the code using a specific toolkit and then test the program on an emulator running on the development machine. Once the application is fully debugged on the emulator, the developer installs it on the actual target device and debugs it further there. Usually, phone manufacturers provide emulators for their devices to be plugged into the development toolkits to allow more accurate emulation of the actual platform. Sun Microsystems' Java toolkits also provide generic emulators.

MIDDLEWARE RESEARCH PROJECTS

We carried out our middleware research in the context of three main projects, whose focus areas are summarized in Figure 1. We did our software development in Java using Nokia Series 60 and Series 80 phones. The platform specifics matter little, as the fundamental distinctions between phones from different manufacturers are minor or nonexistent—the lessons we learned should be equally applicable to most smart phones.

Fuego Core

The Fuego Core (www.hiit.fi/fi/fc) project focused on three main topics:[2] *XML messaging*,[3] *mobile distributed event-based communication*, and *XML synchronization*. Its main contribution has been the integration of current fixed network trends, in particular XML, into mobile computing environments. The project has

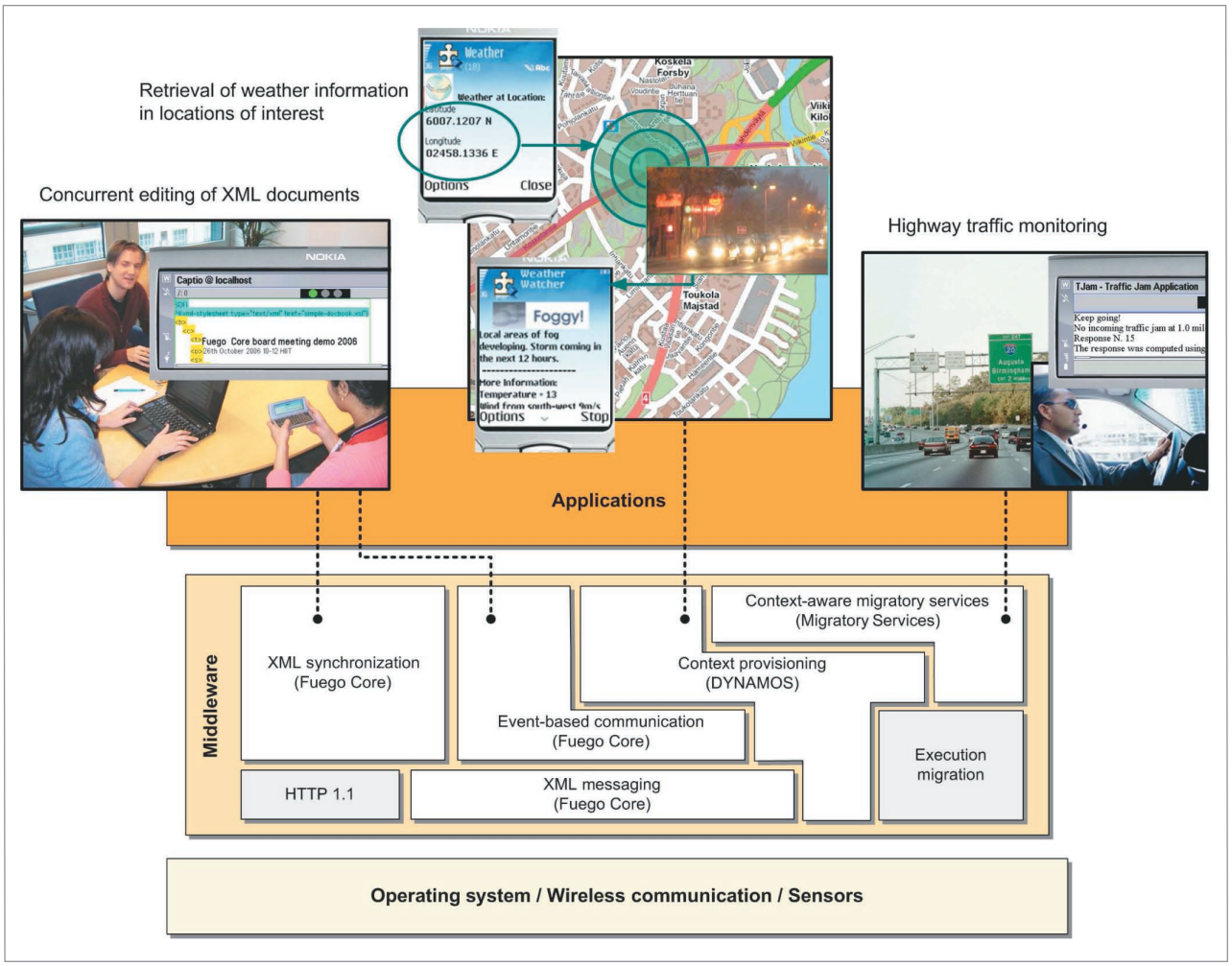


Figure 1. Middleware research projects: research areas, relationships between projects and to external technologies, and examples of applications developed.

also contributed to several international standardization forums, especially the Efficient XML Interchange Working Group (www.w3.org/XML/EXI) of the World Wide Web Consortium, in the area of efficient XML representation.

Using the three Fuego Core components, we built Captio, an XML editor that supports concurrent editing of XML documents by multiple users working on different devices. The application either immediately propagates changes to an edited XML document as events to the other users or synchronizes them later using an XML-aware merging algorithm.

DYNAMOS

The DYNAMOS (www.hiit.fi/fi/dynamos) project focused on *context provisioning* issues. We developed the Contory middleware[4] to support mobile applications that must be aware of both their local context, such as spatial information or network resource availability, and the context of remote entities or physical environments, such as weather information.

Contory can achieve reliable and flexible context provisioning by integrating three alternative sources of context: It can employ local sensors integrated in or connected to the phone, interact with external context infrastructures using event-based communication and XML messaging from the Fuego Core project, or collect sensor information by migrating from node to node in a mobile ad hoc network of sensing devices.

Finally, Contory offers a database abstraction of the sensor-rich environment through an SQL-like programming interface. Using Contory, we built several applications. As Figure 1 shows, WeatherWatcher can retrieve weather-related information in user-specified geographical regions. The context-aware service recommender[5] lets users receive prompt notifications of services available in the surrounding environment as well as generate and share location-specific content.

Migratory Services

The Migratory Services project, in collaboration with the New Jersey Institute of Technology and Rutgers

University, has focused on designing and implementing the *context-aware migratory services*[6] platform to support client-service interactions in mobile ad hoc networks. Unlike a regular service that always executes on the same node, a migratory service constantly monitors its execution context—possibly through Contory's context-provisioning support—and transparently migrates to different nodes in the network to carry out its assigned task. Despite network volatility and dynamically changing execution contexts, migratory services can support continuous and long-running interactions with client nodes.

Using migratory services, we built TJam, a migratory service that predicts traffic jams in a given region of a highway by using only car-to-car short-range wireless communication.

LESSONS LEARNED

Although smart phones are becoming increasingly powerful, their computing power, memory, and network bandwidth are still very limited compared to desktop computers. This was evident during our experimental analysis and led us to conclude that the approach of first developing software for desktop computers and then porting it to phones has few chances to succeed.

Fully addressing the energy problem requires energy-aware mechanisms at all system levels.

Background communication power demands

In spite of their ever-increasing resources, smart phones will always be dependent on a limited energy source. In general, three main elements affect the phone battery lifetime: CPU, display, and wireless communication. In our experimental analysis, we investigated the effects of processing and communication operations on power consumption.

Power consumption. To measure power consumption on phones, we used a multimeter connected in series to a Nokia 6630 phone's battery.[4] Test results showed that power consumption in the idle state is less than 6 milliwatts; with the back light and display turned on, it reaches 76 mW.

Bluetooth's cost lies mostly in the discovery process, with peaks of 292 mW of consumption, while actual communication is relatively inexpensive. For example, transferring 136 bytes costs less than 0.1 joule, which is one-fiftieth of what the discovery process consumes. Having Wi-Fi connected at full signal draws a constant current of 300 mA, which leads to an average power consumption of 1,190 mW. This means that having Wi-Fi connected is more than 100 times more energy-consuming than having Bluetooth in inquiry mode.

Turning on the GSM radio causes additional power consumption that comes in peaks of 450-480 mW every 50-60 seconds. UMTS is also relatively expensive, especially due to the cost for initializing the radio channel, which causes 1-W peaks of consumption for several seconds.

Communication versus computation. We ran a second type of experiment to determine the relationship between energy consumed for communication and for computation.[7] This experiment included both communication of large amounts of data and several seconds' worth of computation. Our results show that on a mid-end phone, the Nokia 7610, transmitting 1 byte over GPRS consumes approximately the same amount of energy as computing for 1.5 ms. We also ran the same experiment using the more powerful Nokia 9500 Communicator. In this case, the byte-equivalent computation time is only 170 µs. While these results demonstrate that compressing data prior to transmission is often beneficial, they also show that the precise gains depend strongly on factors such as device type and network latency.

The fact that wireless communication can be much more expensive than computation, especially on smaller devices, leads us to conclude that sometimes it is better to trade communication for computation. This contravenes the common belief that offloading computation from phones to remote sites saves resources, but, as we have seen, migrating a task and retrieving the results can become very expensive. Further, this high communication cost is an obstacle for many pervasive applications that rely on constantly available background communication for periodic data synchronization or publish-subscribe interactions.

Energy savings. We also experimented with control policies and reconfiguration mechanisms to adjust energy consumption in both Contory and Migratory Services. In our experience, policy-based approaches, despite their popularity, simply are not flexible enough. A priori specification of reconfiguration rules is impossible due to the required close coupling with the platform characteristics. Instead, we believe a learning approach is more promising and potentially capable of identifying the right tricks for saving energy on each specific device.

Fully addressing the energy problem requires energy-aware mechanisms at all system levels. Middleware must be energy-aware in performing network selection, aggregating data, and suspending and resuming applications. The operating system must be energy-aware in managing hardware devices and scheduling data transfers. In addition, the OS must serve as a central monitoring unit for the entire device and propagate contextual information to the upper layers. Finally, it is necessary to coordinate energy-saving actions, resolve potential conflicts, and prioritize tasks whenever possible.

Lightweight protocols and data aggregation

A communication protocol consists of framing and formatting messages. In particular, choosing a suitable data format is crucial for reducing the amount of communication, and thereby energy consumption. Hand-optimized binary formats are beneficial for this, but they often lack extensibility and debugability.

Generic formats such as Java serialization and XML can easily accommodate new kinds of data and are well-specified for debugging, but they can be heavyweight. Java serialization is based on slow reflection, and in our experiments consumed over a quarter of the total time when transferring 1-Kbyte messages in Wi-Fi networks. Recent work on XML encoding, though, shows that it is possible to retain the benefits of XML without sacrificing efficiency.[8]

Two key elements of network performance are bandwidth and latency. To acquire a wide understanding, we ran experiments with both a newer phone, the Nokia E61 (announced in 2006), as well as two older phones, the Nokia 9500 and 7610 (both announced in 2004).

Bandwidth. Our bandwidth measurements over real wireless networks, reported in Table 1, showed that the actual data rate is far from the theoretical maximum even with stationary phones and minimal interference. We could not transfer a sufficiently large amount of data with Bluetooth's streaming Radio Frequency Communications (RFCOMM) protocol to get useful measurements. Attempts to send large amounts of data too quickly invariably crashed the receiving phone's VM. Our best measurement, for a very small piece of data, was approximately 35 Kbps.

Latency. Our latency measurements on the E61 showed that a network round-trip with Wi-Fi is only 10 ms, while EDGE latency is over 300 ms, and even UMTS has a latency approaching 200 ms. EDGE latency on the older 9500 was even larger, on the order of 600-700 ms, comparable to the GPRS latency of 600 ms on the 7610. Moreover, the overhead of the Transmission Control Protocol (TCP) three-way handshake and slow start phase, which are necessary when communicating over the Internet, have their largest effect on small amounts of transferred data. For example, to periodically synchronize the device location with the remote server, our location-based system transfers less than 1,700 bytes of data, and this transfer over UMTS took roughly 770 ms with a deviation of more than 160 ms.

Considering the high latencies of most wireless networks and the TCP overhead, we have two clear recommendations. First, communication connections should be kept open to the extent feasible and reused for further communication. Second, the middleware should aggregate application messages to be sent simultaneously to avoid too many round-trips for sending a sequence of small messages.

Table 1. Wireless network data rates on Nokia E61 and 7610 phones.

Network (phone)	Theoretical data rate (Kbps)	Measured data rate (Kbps)
Bluetooth (E61)	721	—
Wi-Fi (E61)	11,000	1,160
GPRS (7610)	171.2	32
EDGE (E61)	474	207
UMTS (E61)	2,000	328

Data transfer size = 16 Mbytes.

Local processing pitfalls

In our applications, the cost of data communication eclipsed that of local processing and memory accesses. However, this does not mean that local processing can be completely ignored in favor of optimizing communication. Unexpected problems do exist, and lessons learned on desktop computers do not automatically apply to smart phones.

For example, access to flash memory can dramatically increase overall application latency. Our measurements on XML parsing on a Nokia 9500 phone showed that flash memory access decreases performance by an order of magnitude compared to what would be expected by just comparing processor and memory speeds with a desktop computer.

Likewise, the main bottleneck of the Captio XML editor depicted in Figure 1 turned out to be refreshing the screen. This surprised us—because each key press was echoed through a remote server over Wi-Fi, we expected the messaging to determine the application responsiveness, but it turned out that just repainting the screen takes 500-800 ms,[9] well above the latency for communicating over Wi-Fi.

While CPU usage is not usually a bottleneck in the applications considered here, it cannot be neglected. As indicated previously, CPU usage due to Java serialization can highly affect an application's communication latency. Moreover, some specific applications, such as those that do cryptographic processing, can spend a large proportion of their total time using the CPU. For example, we measured signature verification to take over 2 seconds on the Nokia 7610 phone.[7]

No connectivity, no sensor access

The need for sensor information on mobile devices has grown with the advent of context-aware applications such as tourist information services, healthcare monitors, sport trainers, and navigators. Sensor devices range from common GPS receivers for positioning information to advanced accelerometers for activity recognition,

environmental sensors for weather estimation, and biosensors for health monitoring.

Proper support for context-aware applications requires these sensor devices to be available to the phone and be able to produce real-time measurements. Currently, few phones integrate sensors; Nokia announced the N95, the first smart phone to include integrated GPS, in late 2006. The more common alternative, external sensors, are usually connected via Bluetooth with a Bluetooth-based programming API. Yet, apart from Bluetooth GPS devices, most Bluetooth-based sensors can provide only logs of sensor data, not real-time measurements.

Based on our experiences, sensors also are not yet sufficiently reliable, and communication problems sometimes hamper the adoption of remote context servers. In 2005, we organized a sailing regatta in which nine sailboats used the location-based application developed in the DYNAMOS[5] project on Nokia 6630 phones with Bluetooth GPS receivers. The application communicated periodically with a location server using the 2G or 3G network to send location updates and retrieve location-based content. Several types of disconnections occurred. The Bluetooth connection to the GPS device went down once per hour. Whenever the 3G connection was active and the phone had to make a handover to 2G, it switched itself off. We had to severely optimize the protocol for communicating with the location server to reduce energy consumption and keep peak power consumption below the limit over which the phone would switch itself off.

Proper support for context-aware applications requires sensor devices to be available to the phone and be able to produce real-time measurements.

Finally, deploying sensor-based applications remains complicated. Because there is no standard API for managing and controlling connected sensors, it is necessary to program each sensor differently.

For Java ME, the Mobile Sensor API—Java Specification Request (JSR) 256—is currently under development, but it will take time for this work to be completed and become available on actual devices. Meanwhile, middleware can provide a temporary patch by unifying different sensor data protocols. Even in this case, designers must be careful not to bloat the middleware by supporting too many types of sensors.

Ad hoc networks not yet spontaneous

Wi-Fi and Bluetooth, now commonly available on smart phones, make ad hoc networks feasible. Wi-Fi offers a more flexible solution and suits applications requiring large data transfers or operating in complex network topologies. Bluetooth is better suited to applications that open sporadic short-lived connections to other devices and operate in more stable configurations.

Configurability and reliability. Our experiences with using ad hoc networks of phones revealed configurability and reliability problems. First, configuring an ad hoc network of phones is not straightforward. For example, with Nokia 9500 phones, an ad hoc network becomes active when a phone first connects to the Internet access point for that network. In our tests, a device could not connect to an IAP without explicitly sending data out—opening a socket and listening on it was insufficient. Moreover, even when an address is assigned to the listening device, other nodes do not have a Bluetooth-like mechanism to discover this address. Rather, the application developer must implement service discovery as well.

Wi-Fi and Bluetooth limitations. Currently, support for Wi-Fi-based ad hoc networks on phones is woefully lacking. The developer must implement all basic functions such as routing, neighbor discovery, and message aggregation. Further, only Internet protocols are available for programmers, so these must be User Datagram Protocol (UDP)-based systems.

Although most Bluetooth functionality and protocols are available directly to developers, Bluetooth-based ad hoc networks present several other limitations. In a Bluetooth piconet of phones, the master phone can establish multiple links to at most seven phones. The master-slave switch is generally not supported on phones, preventing the creation of scatternets: It is impossible to route messages across piconets. In addition, the Bluetooth stacks on the phones we experimented with were relatively unstable, often crashing the VM.

Performance. Finally, performance is a serious problem with ad hoc networking on phones. Over a one-hop Wi-Fi network, we measured a round-trip time of 340 ms for a 1-Kbyte message, and over Bluetooth an RTT between 450 and 600 ms. Frequent interference problems occurred in places with a medium or high density of wireless devices, rendering Bluetooth mostly unusable in practice.

Awkward asynchronous programming

Pervasive applications are essentially reactive and must constantly monitor their surrounding environment and act upon changes in it. Asynchronous programming models and languages are therefore preferable.

Support for asynchronous programming in Java ME is poor. The programmer must create a separate monitoring thread for each potential event to be captured. Each thread then blocks waiting for its specific event and becomes active when the event happens. If several monitoring threads become active concurrently, they must compete for CPU time, leading to a severe degradation in performance.

In addition, multithreading on smart phones is not recommended, especially if the number of threads can increase without bounds. As smart-phone OSs are not specifically designed for good multitasking performance, context switches can be very expensive, and all threads, even those not running, will consume application memory.

A further problem with Java is its high level of abstraction. Normally, a Java program handles I/O in a uniform manner, independently of whether it comes from a network or local storage. With pervasive computing, this model breaks down, as local I/O can often be expressed synchronously, but network I/O must always be asynchronous. Thus, the correct level of abstraction for I/O must distinguish between high- and low-latency operations, which makes even the Unix model of descriptors unsuitable.

Symbian C++ offers better asynchronous programming facilities. The active objects system provides a centralized event handler that uses registered observers to handle the events using only a single thread. While the active object system is not the easiest to use, its design is sufficient for pervasive applications.

Pervasive applications are essentially reactive and must constantly monitor their surrounding environment and act upon changes in it.

Finally, both Java and C++ are languages fundamentally built around synchronous processing, and adding asynchronous functionality requires attention from the programmer. If the languages were initially designed for a concurrent-processing environment where nodes are expected to be unreliable, it would be possible to provide a more adequate reactive programming model.

Laborious software development process

Programming smart-phone platforms introduces additional limitations that make application design and development on phones different from, and usually more laborious than, the standard process.

Concurrent programs. Java ME does not support the execution of multiple independent programs concurrently. When running, the Java VM monopolizes the phone and must terminate a running program to allow another to run. This impacts application design, as every component must be written as a library to link into the running program.

The usual solution to let part of a program run as a separate component is to implement its functionality as a C++ program that listens on a local socket. The main program will then be able to communicate with the other program as if it were a network server. This is inefficient and also necessitates designing the protocol through which the two programs communicate.

Memory usage. Another important consideration in designing a phone application is its memory usage pattern. Current smart phones, even high-end ones, usually do not have more than 10-20 Mbytes of memory for a running application, so designers must completely rethink allocation behavior and program composition. This is especially a concern with Java, where the ME platform's similarity to the SE version can encourage programmers to port existing software with minimal effort. Tommi Mikkonen[10] provides advice on designing programs for phones that basically amounts to forgetting many established object-oriented design principles.

Debugging applications. The debugging process is also rather complicated and calls for running applications both on an emulator and the target devices. So-called emulators do not really emulate an actual phone but essentially just run the phone code on the development machine, thus they cannot reliably provide an accurate model of the actual target device. Moreover, debugging must often be done directly on actual devices simply because emulators do not support some features. This is particularly true when debugging multiphone applications, as some environments do not support running two emulators or provide network communication between them.

Target device. The actual target device presents an additional complication. Different phones have various OS versions, VMs, and so on. This means that to build a truly robust application, the developer must test the program code on all target devices, not just one of them.

RELATED WORK

Numerous other research projects have focused on middleware for pervasive computing. However, most of the proposed systems were built using technologies that smart phones do not support or that require excessive resources unavailable on phones. Only a fraction of this research work has targeted practical development on phones.

Smart Messages

Rutgers University has undertaken several research activities in this area. Of particular interest is the Smart Messages[11] platform for cooperative computing in mobile ad hoc networks. With Smart Messages, the application's execution is sequentially distributed over a series of nodes using execution migration. Nodes are named by properties and discovered dynamically using application-controlled routing. We used this platform to program mobile ad hoc networks in Migratory Services and Contory.

CAPNET

At the University of Oulu, the CAPNET (context-aware pervasive networking) research program

(www.mediateam.oulu.fi/projects/capnet) has focused on designing a component-based middleware offering functionality for sensor monitoring, context recognition, context-based reasoning, application adaptability, service discovery, and so on. For example, CAPNET researchers developed a radio-frequency-identification-based application for requesting services. When a user activates an RFID tag via his phone, this generates a context event, and the application reacts either by providing the requested service or by forwarding the request to an external component.

ContextPhone

At the University of Helsinki, the Context project developed ContextPhone,[12] a software platform providing open source C++ libraries and source code components to support context-aware applications on Nokia Series 60 phones. For example, ContextContacts is a prototype application that permits visualizing the current contextual state of all contacts listed in the phonebook.

Wearable computing and HCI

Finally, research on wearable computing and human-computer interaction is relevant to our work. Carnegie Mellon University's SenSay[13] prototype relies on sensor data and user information to infer the user's status and situation and adapts its behavior accordingly. For example, it can inform a remote caller when the user is unavailable or hide incoming calls when the user is busy, as well as increase the ringtone volume when necessary. SenSay employs light, motion, and microphone sensors placed on the human body that communicate with a central sensor box worn at the waist.

In the same research vein, Jani Mäntyjärvi and colleagues[14] proposed a touch-detection system for mobile devices. The system uses two sensor pads placed on the phone, and skin impedance measurements to detect the presence of a hand or other objects.

When comparing middleware for desktop computers to middleware for smart phones, energy management is the most prominent difference. Because achieving good application performance usually also leads to higher energy consumption, good middleware for smart phones is not what can provide all conceivable services, but rather what understands the acceptable tradeoffs between level of performance and resources needed and can adjust its behavior accordingly.

However, the middleware layer is too high for fine-grained resource management, thus energy management needs help from lower layers too. The OS is ultimately responsible for the resources, and it can distribute energy to competing tasks based on user preferences and task priorities. Going even lower, smart batteries that provide information on current capacity, drain rate, and voltage can help configure the entire system's energy profile.

Consider, for example, how this cross-layer energy management applies to network communication. The middleware is responsible for making the communication protocol compact and deciding what to transfer. The OS must time the data sending properly to avoid continuous manipulation of the radio interface. And at the lowest layer, the physical and link layer protocols need to be designed to permit low energy consumption for communication.

Middleware on smart phones must also consider an inherent property of most pervasive applications: their need to constantly react to external events. Middleware can help meet this goal by acting as a framework instead of a library component, but developers must nevertheless rethink existing programming languages to fully enable programming pervasive applications.

In addition to addressing these core issues, researchers building middleware for smart phones should test their solutions on actual devices. While programming phones is a tedious task due to limitations and software bugs, the behavior of real devices is too variable and unpredictable to be captured through simulation alone.

Acknowledgments

The authors thank the members and funders of the Fuego Core, DYNAMOS, and Migratory Services projects. They also thank Cristian Borcea and Jussi Kangasharju for their comments on an earlier draft of this article.

References

1. G. Roussos, A.J. Marsh, and S. Maglavera, "Enabling Pervasive Computing with Smart Phones," *IEEE Pervasive Computing*, vol. 4, no. 2, 2005, pp. 20-27.
2. S. Tarkoma et al., "Fuego: Experiences with Mobile Data Communication and Synchronization," *Proc. 2006 IEEE 17th Int'l Symp. Personal, Indoor and Mobile Radio Communications* (PIMRC 06), IEEE Press, 2006.
3. J. Kangasharju, T. Lindholm, and S. Tarkoma, "XML Messaging on Mobile Devices: From Requirements to Implementation," *Computer Networks*, vol. 51, no. 16, 2007, pp. 4634-4654.
4. O. Riva, "Contory: A Middleware for the Provisioning of Context Information on Smart Phones," *Proc. ACM/IFIP/Usenix 7th Int'l Middleware Conf.* (Middleware 06), LNCS 4290, Springer, 2006, pp. 219-239.
5. O. Riva and S. Toivonen, "The DYNAMOS Approach to Support Context-Aware Service Provisioning in Mobile Environments," *J. Systems and Software*, vol. 80, no. 12, 2007, pp. 1956-1972.
6. O. Riva et al., "Context-Aware Migratory Services in Ad Hoc Networks," *IEEE Trans. Mobile Computing*, vol. 6, no. 12, 2007, pp. 1313-1328.

7. J. Kangasharju, T. Lindholm, and S. Tarkoma, "XML Security with Binary XML for Mobile Web Services," *Int'l J. Web Services Research,* vol. 5, no. 3, 2008, pp. 1-19.
8. G. White et al., eds., "Efficient XML Interchange Measurements Note," World Wide Web Consortium working draft, 25 July 2007; www.w3.org/TR/exi-measurements.
9. T. Lindholm and J. Kangasharju, "How to Edit Gigabyte XML Files on a Mobile Phone with XAS, RefTrees, and RAXS," to appear in *Proc. 5th Ann. Int'l Conf. Mobile and Ubiquitous Systems: Computing, Networking and Services* (MobiQuitous 08), IEEE CS Press, 2008.
10. T. Mikkonen, *Programming Mobile Devices: An Introduction for Practitioners,* John Wiley & Sons, 2007.
11. N. Ravi et al., "Portable Smart Messages for Ubiquitous Java-Enabled Devices," *Proc. 1st Ann. Int'l Conf. Mobile and Ubiquitous Systems: Computing, Networking and Services* (MobiQuitous 04), IEEE CS Press, 2004, pp. 412-425.
12. M. Raento et al., "ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications," *IEEE Pervasive Computing,* vol. 4, no. 2, 2005, pp. 51-59.
13. D. Siewiorek et al., "SenSay: A Context-Aware Mobile Phone," *Proc. 7th IEEE Int'l Symp. Wearable Computers* (ISWC 03), IEEE CS Press, 2003, pp. 248-249.
14. J. Mäntyjärvi et al., "Touch Detection System for Mobile Terminals," *Proc. 6th Int'l Symp. Mobile Human-Computer Interaction* (Mobile HCI 04), LNCS 3160, Springer, 2004, pp. 331-336.

Oriana Riva *is a senior researcher in the Department of Computer Science of the Swiss Federal Institute of Technology, Zürich (ETH Zürich). Her research interests include mobile computing, middleware for resource-constrained mobile devices, vehicular networks, and sensor networks. Riva received a PhD in computer science from the University of Helsinki. Contact her at oriva@inf.ethz.ch.*

Jaakko Kangasharju *is a postdoctoral researcher in the Department of Computer Science and Engineering at Helsinki University of Technology. His research interests are middleware, application development, and XML use on small and mobile devices. Kangasharju received a PhD in computer science from the University of Helsinki. He is a member of the IEEE and the ACM. Contact him at jkangash@cc.hut.fi.*