Component Based Invisible Computing

Alessandro Forin, *MSR*[*], Johannes Helander, *MSR*, Paul Pham, *MIT*, Jagadeeswaran Rajendiran, *Duke*

Abstract-- **MMLite is a modular system architecture that is suitable for a wide variety of hardware and applications. The system provides a selection of object-based components that are statically and/or dynamically assembled into a full application system. The virtual memory manager is optional and is loaded on demand. Communication with remote peers uses XML/SOAP and standard web services. Components can be easily replaced and reimplemented. Componentization reduced the development time and led to a flexible and understandable system.**

MMLite emphasizes real-time and provides a selection of schedulers, such as a feedback constraint-based scheduler. The scheduler is a selectable component and can be replaced so that different policies can be implemented. Minimal latency to interrupts and preemption provides the scheduler with maximum freedom to schedule tasks according to the chosen policy.

MMLite is efficient, portable, and has a very small memory footprint. It runs on several microprocessors, including two VLIW processors. It has been used on multimedia and gigabit ethernet cards, sensor devices, handheld games, and various embedded development boards.

1 INTRODUCTION

As semiconductor technology becomes more mature and inexpensive it becomes feasible to add computing and/or communication capabilities to many devices that used to be mechanical or analog, and to come up with new devices entirely. Computing that enhances existing everyday devices and makes them smarter without requiring extra human interaction is called *invisible computing*. In this domain the computer is not the main focus but rather the device itself or the specialized function it performs.

Invisible computing differs from personal computing mainly because the user interface is not screen and keyboard based and resources (such as energy, memory, bandwidth, budget) are often severely restricted. While a PC or workstation can use a general-purpose operating system (a collection of commonly needed features) an invisible computer can seldom afford such luxury because of resource constraints. The software must instead be tailored to the specific application.

Invisible computing is also slightly different from traditional embedded computing in that the devices are most often communicating with each other and/or with general-purpose computers (e.g. via low-power wireless) and have limited power supply.

Limited resource, semi-intelligent devices found in invisible computing environments can perform rudimentary tasks autonomously. It is the ability to communicate with other invisible devices that gives them added capabilities, such that the value of the whole system is greater than the sum of its parts.

When the small devices have the ability to communicate with PCs or other "big machines" in addition to other peers, it is possible to leverage the advantages of invisible computing in desktop computing and vice versa. PCs can provide backend processing for small devices, and small devices can extend the reach of a traditional PC further into our everyday lives. For example, a small device can access a database on a web server; aPC can provide a user interface for examining and analyzing sensor data in a home.

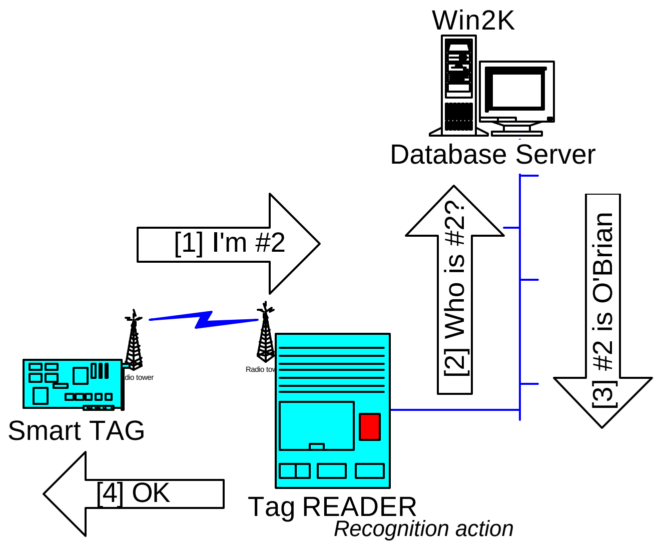

Figure 1: SOAP messages in a smart tag scenario

Consider the system in Figure 1. A person carrying an active badge approaches the entrance of a secure building, which is equipped with a tag reader. Access can be granted based exclusively on information exchanged by the tag and the reader: a cryptographic key or a challenge protocol. Additional functionality is available if the reader or the tag has access to services on the net. The tag might obtain its key from a web authentication service such as Passport. The reader might validate with the use of a database server. The database could also provide a picture or other additional

[*] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
Email: {sandrof,jvh}@microsoft.com

information such as a personalized greeting and other preferences to smart devices inside the secure building.

We have experimented with using generic XML based protocols such as SOAP [19] for communication between devices and with internet services. Our preliminary results are that this is both feasible and that it provides extra flexibility in composing a distributed system. XML based protocols are a good fit for reusable component software. Some performance optimizations can be beneficial for XML on small devices.

Devices of interest to our system work include:

- Embedded control systems, including consumer devices, intelligent sensors and smart home controls.
- Communication-oriented devices such as digital cell phones and networking infrastructure.
- Programmable peripherals and microcontrollers.

In all these cases, the general-purpose platform approach is either not applicable, or it is prohibitively expensive. The microprocessor might be a DSP, a VLIW, or a micro-controller; the memory budget is severely restricted; there might be no MMU; the network connection might be sporadic; and real-time is essential.

Current operating systems are either inflexible, big, lack real-time support, have complex hardware requirements, or are so specialized that good development tools are unavailable and code reusability is low.

In this paper we discuss MMLite [8], a system architecture that is suitable for a wide range of applications. Our strategy is to build a system out of minimal but flexible components. Instead of mandating a fixed set of operating system services and hardware requirements, we provide a menu of well-defined software components that can be chosen to compose a complete system depending on hardware capabilities, security needs, and application requirements. Components can be selected at compile time, link time, and run-time. Components can be transparently replaced while in use, via a mechanism we call *mutation*.

The process of targeting parts of the menu of available capabilities and components to a specific application is called *specialization*. Logically it consists in partially evaluating the entire software base against all possible executions of the application, identifying what is (mostly) a constant and eliminating the unused parts. When something that was assumed constant changes, the system must be dynamically respecialized. This is done by loading new components and creating new objects when possible and by mutating existing objects when necessary. Specialization reduces footprint and must be done aggressively in the systems of interest to us.

Componentization makes it easier to change an implementation of a component without affecting the rest of the system as long as the interface of that component is unaffected. Minimalism forces the system to be understandable and adaptable. Software components, when possible, are not tied to a particular layer of the system, but can be reused. For example, the same component that implements the system physical memory heap is used to provide application heaps over virtual memory. We have componentized the system more aggressively than any previous operating system. This includes the virtual memory system, IPC, and the scheduler in addition to filesystems, networking, drivers, and security policies.

The rest of the paper is organized as follows: Section 2 describes the system architecture. Section 3 describes the implementation of a few major components. Sections 4 and 5 give examples of applications where MMLite has been used and the status of the system. Related work is presented in section 6, and conclusions in section 7.

2 ARCHITECTURE

C++ and Java provide objects at a very fine granularity level, and they are very successful with application programmers. Unfortunately, both languages confine their objects to a single address space. Object Linking and Embedding (OLE) [2], CORBA [14], and other similar systems extend objects across address spaces and across machine boundaries. OLE seamlessly integrates independently developed components. When editing an Excel spreadsheet inside a Word document it is in fact the Excel process that operates on objects inside of Word's address space. Unfortunately, OLE only works for user mode applications. MMLite takes an "objects everywhere" approach, and extends object-orientation both across address spaces and across protection levels.

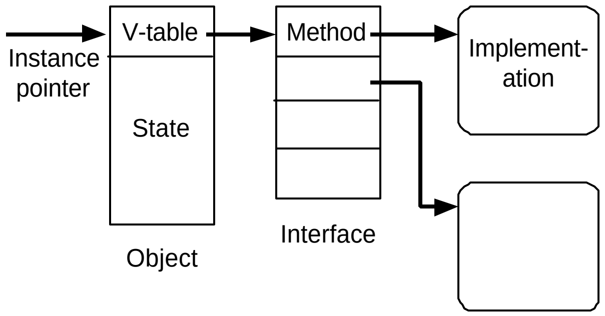

Figure 2: A run-time object representation.

2.1 *Component Object Model*

MMLite components contain code and other metadata for classes of objects. When a component is loaded into an address space it is instantiated. The instantiated component creates object instances that communicate with other objects, potentially in other components. The objects expose their methods through Component Object Model (COM) [2] interfaces. MMLite uses the COM mechanism but does not implement the libraries or APIs of regular COM. MMLite objects can be made available to other components by registering them in a namespace. Namespaces are similar to filesystem directories but are not limited to just files.

The object model enables late binding, version compatibility and checking, transparency through proxies, and cross language support, and it is reasonably lightweight and efficient. As in many object systems each object has a method table pointer and a reference count. Each call adds one indirection for fetching the actual method pointer.

MMLite component implementations are rarely aware of the system layer in which they are intended to run. The same code can be used in different address spaces or contexts and can be recursive (e.g. an application heap can depend on a system heap that runs the same code). A filesystem can be applied to a file provided by another filesystem as well as to one provided by a disk driver. A heap can be applied to any memory: physical memory, memory allocated from another heap, or memory provided by a virtual memory manager. The loader loads modules into any address space

The recently introduced C# language embodies much of the same software engineering thinking that went into the MMLite design. In addition to extensive language support for componentization this new language dispenses with many annoying trivialities like reference counting and metadata generation.

2.2 Namespaces

Namespaces are used to let applications gain access to objects provided by other components. A namespace is like a filesystem directory tree, except it can hold any kind of objects, not just files. Namespaces can themselves be implemented by different components, including a filesystem that exports its directories as sub-namespaces, and files as registered objects. Namespaces can be registered into other namespaces, extending the directory tree. Location transparency of all objects automatically makes namespaces distributed. Namespaces can be filtered for access control or for providing different views to different applications. There is no limit as to the number of namespaces. A component can gain access to its root namespace through a call to CurrentNamespace(). In a minimal system all applications share the same (boot) namespace.

We implemented a demand-loading namespace that supports the following new programming model. The *main()* entry point for an image is a constructor that returns an object. When an application tries to bind to a name that does not exist, the namespace invokes the loader, which looks in a filesystem namespace for a component with the given name and loads and instantiates it. The loader then invokes the component's entry point, registers the resulting object in the namespace, and returns it to the application. The returned object is often a factory (constructor) for creating other objects. When the application releases its last reference to the component the namespace can unload the component or choose to keep it cached.

The demand-loading namespace is also used to hide configuration details. When a component is configured to be statically built into the boot or ROM image, the namespace will call the entry point directly without any loading based on a built-in table. This way a user of the component does not need to know whether a component was built-in to the system or where it might get loaded from.

2.3 Selection of System Components

What components should be part of a deployed system depends on the applications themselves and their interface requirements, application memory requirements, security requirements, and the target hardware capabilities. Flexible loading of modules was an important design goal for our system. The loading of components can be deferred until they are actually used by an application. Device drivers and run-time services typically fall into this category. Others such as virtual memory for untrusted applications can be loaded just prior to running an application. Most services will terminate themselves when they are no longer needed. The structure of the system might change radically during execution in response to external events.

Drivers and virtual memory cannot be used when the hardware to support them is not present. An application that tries to use them will look them up in the demand-loading namespace. An error is returned, because either the driver is absent or it returns a NULL pointer instead of a valid object during initialization.

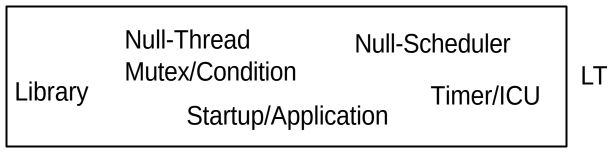

Figure 3: A minimal system configuration. Components are loaded at link time (LT).

Figure 3 shows a minimal system configuration that can be used in a watch. Figure 4 shows a larger system configuration that can be used in a card reader. Applications can run within the physical address space, within a separate address space, and within a virtual machine. The IPC system uses the network and virtual memory mappings as its transports.

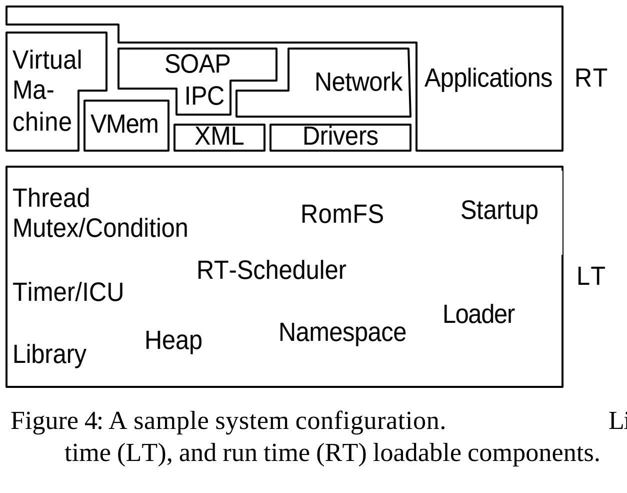

Figure 4: A sample system configuration. Link time (LT), and run time (RT) loadable components.

2.4 XML Based Configuration

Global interface and component information is kept in a set of XML data files. The data contains all the interface specifications both for programmatic and for human use. A tool processes the data and produces online reference manuals, C header files, XML typeinfo for marshaling, code skeletons, and other ancillary files. Component descriptions are cross-referenced and used for generating configurations and for simple analysis. Once all the necessary information is present in one place in a machine readable form more sophisticated analysis is possible; this is a topic of future research. However, having an online manual that is always consistent with the code has already proven very useful.

2.5 *Virtual Memory*

Unlike most existing operating systems, MMLite does not make the support for virtual memory an integral part of the system. The system can function with or without it, and it executes the same binaries. The virtual memory manager is a component like any other; it can be loaded and unloaded dynamically on demand, permanently built into the system, or completely left out.

2.6 *XML Based Communication*

The emerging lingua franca of programmable web services is XML. To make invisible computing devices useful as services to big machines and web services accessible from the small devices it seems logical to speak the common language. It might be argued that specialized protocols are more efficient, but in many cases they turn out to be equally or more complex, and they mandate complicated and costly proxy or gateway machines. We implemented SOAP for small devices and concluded that the resulting code is reasonably small and efficient and that there is no need for special purpose protocols. It is actually a great savings in complexity to be able to communicate directly with the relevant services and peers in a common protocol. Some optimizations are desirable, however, and they would not benefit just small devices:

- Unnecessary protocol layers can be eliminated. For example most SOAP implementations run over HTTP, which is about as much code as SOAP itself.
- Textual XML representation incurs large parsing and formatting overhead. A binary representation can save a lot of CPU and consequently energy.
- Network layers can be split into further sublayers so that some of the sublayers become candidates for elimination.
- Specialized compaction, such as separating constant parts from variable parts of a message can significantly reduce message sizes.

3 SYSTEM COMPONENTS

The menu of base components includes the following. All of them have been implemented and tested. For sizes, see Table 1 in section 5.

- **Heap:** Physical memory management. Two regular implementations and a temporary heap are provided.
- **Loader:** Enables loading new components into the system. Several image formats are supported.
- **Support Library, Machine Initialization**
- **ISO C Library**
- **Timer and Interrupt Objects:** A driver for the timer chip is used by the scheduler to keep track of time and for thread pre-emption. The driver dispatches interrupts to registered interrupt service routines, which can be implemented by other components.
- **Scheduler:** A policy module that determines which thread should run at any given time.

  Low-level management of blocking and switching between threads is handled by the thread and synchronization components, which call into the scheduler, possibly passing callback functions as arguments.

  Five schedulers have been implemented: the null scheduler, a simple round robin scheduler, a constraint based real-time scheduler, a simple periodic scheduler, and another independently implemented real-time scheduler [1]. The null scheduler is for systems that use only one thread. Constraint scheduling is for consumer real-time applications and is described in [12].
- **Threads and Synchronization:** Basic thread support and synchronization primitives. Threads can block on mutexes and conditions. They can inform the scheduler of their time constraints, but these calls will fail if the scheduler is not a constraint scheduler. The constraint scheduler performs priority inheritance when threads block on mutexes.
- **Namespaces:** A simple boot namespace where applications register objects. A namespace that cooperates with the loader in demand-loading and caching of components. A namespace, used for displaying the status (e.g. running threads) and performance parameters (e.g. execution times) of a system during development. Filesystems are also loadable namespaces.
- **Filesystems:** Used to load additional components during run-time. We implemented *RomFS* for read-only in-memory images (arbitrary files and the system can be merged into one image) and *FatFS* for reading/writing disks. *NetFile* is a simple network filesystem client built on top of sockets.
- **Network:** TCP/IP
- **Startup Program.**
- **Atomic Queues and a DMA manager** are useful to device driver writers.
- **Virtual Memory Implementation:** An optional loadable and unloadable virtual memory manager.
- **A Minimal Web Browser:** A graphical user interface component, based on an LCD and push-buttons. The browser optionally also implements Visual Basic scripting.
- **Games:** Games like Snake or Doom offer another approach to user interfaces.
- **Ciphers:** Encryption provides the basic building blocks for security and authentication.

3.1 Web Service Components

These components make it possible for an MMLite device to communicate with web services or to act as one. They can be composed in many ways.

- **Tokenizer:** Reads text data from a network stream and splits it into units of text, based on context. The tokenizer is used by the HTTP server and SAX [18] parser. It facilitates processing of network data while it is being received, meaning that the entire request does not have to be present at once. Footprint is therefore reduced.
- **SAX Parser:** Parses XML data as it is being received and calls event driven handler functions through a COM interface. One implementation of the SAX handler interface formats XML for reply messages.
- **BAX Processor:** Deals with pre-tokenized XML [BAX stands for *Binary API for SAX* and is work-in-progress]. This component is similar to the SAX parser but handles binary XML thus saving significant processing that is otherwise associated with textual data. A standard interface and format has not yet been defined. A standard representation will be needed for making binary XML useful between machines.
- **HTTP Server:** Handles simple HTTP requests such as reading and writing files. URLs map easily to MMLite namespaces. The HTTP server also allows sending SOAP messages embedded into HTML pages, in which case the HTTP server delegates the work to the SOAP marshaler and SAX parser.
- **SOAP COM Marshaler:** Provides automation for accessing COM objects through SOAP. Note that SOAP messages can be handled directly as messages as well as marshaled into COM objects.

  On the server side this is a handler for SAX or BAX. Any MMLite object can be accessed through a URL that starts with */SOAP* as long as the marshaler has access to the corresponding typeinfo (XML based metadata).

  On the client side proxy objects are automatically generated corresponding to the typeinfo. When a proxy method is called, it is marshaled into a SOAP request and the reply is then processed using SAX or BAX.

  Aside from timing and potential communication link failures, a remote object call through a proxy looks exactly the same as a local method call directly to the actual object.

4 APPLICATIONS

MMLite has been used in various devices, smart I/O cards, and several development boards. This section looks at some of those scenarios.

4.1 Devices

We built a simple hardware platform for experimenting with sensors and devices for invisible computing. We used an Atmel AT91FR4081 ARM-based microcontroller, an (optional) small LCD, battery charger, buttons, and a piezo-electric buzzer on a low power (40mW at 2.8V, full speed) relatively small size (5 by 7 cm) processor board. An add-on board provides an accelerometer, magnetometer, gyro, and a 40kb/s radio. The add-on board consumes 130mW when transmitting and sensing, 70mW in standby. A two-player game can be played between two devices over the radio, using the buttons and the accelerometer for input. The entire system runs in the 128KB on-chip RAM.

A prototype “smart tag” and “reader” using XML SOAP messaging and web services is running on ARM-based Cerf-boards using 802.11 wireless [Figure 1].

MMLite was used in a personal area network hub, or BodyHub, in a joint demo with the Portolano project [5] in October 2000. The hub connected instrumented biology laboratory equipment to back-end services by bridging BodyCom [15] to 802.11 with filtration and augmentation of the sensory data.

4.2 Smart I/O Cards

MMLite was used in a prototype TCP Winsock Direct Path [21] implementation as the operating system on a Gigabit Ethernet card, with the host machine running Windows NT. The CPU on the card was rather slow and did not have interrupts, a cache, or multiplication instructions. Yet MMLite was very useful as a base system for writing the data pump that controlled packet transmission and reception, as well as DMA directly into the host’s user-mode application buffers, in collaboration with the Windows NT virtual memory system. To optimize execution we used a new loader feature that allowed marking code and data *hot* in a program so that it would be put in a special section. The loader then automatically loaded this section into the processor’s on-chip 4KB of fast SRAM. The resulting code was able to sustain 118 MB/s on a PCI bus that had a capacity of 120 MB/s as measured by a PCI analyzer. The round trip time for sending a message from a user mode NT application on one machine to a user mode application on another machine and receiving a reply message back was 18 µs.

Equator Technologies independently ported MMLite to its proprietary VLIW media processor, and used it to run multiple simultaneous multimedia applications, including MPEG2 Decode, AC3, 3D Graphics (D3D) and telecommunications.

The MMLite system was used at Microsoft in a prototype Talisman [17] 3D graphics and multimedia card. This involved supporting DirectDraw and Direct3D, DirectSound and a wavetable based software MIDI synthesizer implementing the DLS level 1 specification. On a 90MHz PC we measured in excess of 7,800 RPC/second between a user mode Windows NT application and an MMLite component, with time predominantly spent in NT. An interactive game (Doom) can also run directly on the board, along with a number of other demonstration programs, validation tests and regression tests.

Figure 5 depicts the structure of the MIDI synthesizer components in the Talisman card’s audio subsystem. The

SYNTH thread runs periodically and when new MIDI data arrives. It produces an output buffer of 44kHz, 16bit, stereo audio samples. The MIXER thread also runs periodically, and expects to find a new buffer available on each of its input channels. If an input buffer is not present, a buffer of silence data is used instead. Buffers contain data in any number of formats; the mixer adapts frequencies, stereo/mono, and number of bits per sample. The result of the mixing is given to the AUDIO driver, which starts the DMA to the D/A converter. Completed buffers are returned to the mixer directly by the interrupt routine. The Atomic Queues component make data movement easy as no locking is required between drivers and/or the interrupt code.

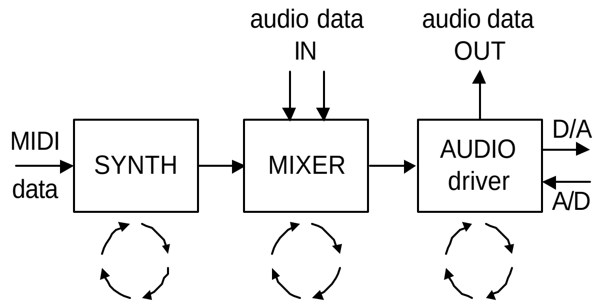

Figure 5: MIDI synthesizer components.

All of the audio components use time constraints and can without interference run together with DirectDraw and other applications. The overall CPU load when running these components under MMLite on an 80MHz TriMedia processor is 7% for the SYNTH component, and 3% for the remaining audio components.

5 STATUS

The system currently runs on the i386, ARM (many versions), Philips TriMedia and Equator MAP1000 VLIW processors, as well as H8, MIPS, and 68k.

Heap1	2635	Boot NS	1265
Heap2	3420	Dload NS	512
PE loader	4661	RomFS	1417
Library	3799	FatFS	8229
Machdep	2086	NetFile	6944
Timer	1205	Startup	118
ICU	1005	Network	84832
Null-sched	316	XML/SOAP	16KB
RR-sched	599	HTTP server	12KB
RT-sched	1228	AtomicQueue	415
Thread	426	VMem	17712
Synchro	1090	Doom	285696

Table 1: i386 components and their binary sizes in bytes.

The size of the minimal system in Figure 3 is 10KB on i386, excluding a boot stack. The size of the base system (the LT box in Figure 4) is 26KB on i386, 20KB on ARM. Table 1 lists the sizes of the components of Section 3.

6 RELATED WORK

OSKit [7] shows how a base set of system components can be composed in different ways to build an operating system kernel. The work is mostly concerned with reusing existing device drivers and Unix code and does not attempt to componentize the core of the operating system, nor does it concern itself with applications.

Chorus [16] is the only system we know of that can be configured to use either a page-based or a segment-based VM system. MMLite is the first one that can run with or without VM, and dynamically load and unload it — unless, of course, we look at MS-DOS in a very twisted way.

Rialto [11] shows how the COM model can be implemented in the presence of VM, and argues for a unified programming model that is independent of the privilege issue (no user versus kernel distinction) [4]. We show how those same principles are beneficial in scaling down a system to cope with resource poor domains.

Windows for Smart Cards [20] implements an event driven programming model, including the messaging standard ISO 7816. It can be used in a subset of the MMLite application space, but the programming model and other limitations are probably not well suited for most applications outside the smart card domain. Similar models are also implemented by TinyOS [9], which requires a special-purpose network protocol, and by SPINE [6]. A special-purpose network protocol necessitates proxy computers for communication with the rest of the world, departing from the goal of ubiquitous communication. SPINE offers extensibility through Modula3. Windows for Smart Cards, like MMLite, has Visual Basic scripting.

CORBA [14] forces all calls to go through the object request broker, thus penalizing the local case. Real-time support in CORBA is still at the research stage [22].

Componentization and location independence has also been studied in the context of file systems and network protocols [13] and in a number of existing embedded systems, such as pSOS [10]. In a typical embedded system there is no loader; components can only be chosen at static link time when the load image is built. Services are extremely limited, sometimes just to the scheduling component. The number and priority of threads might have to be specified statically as well.

Modularity has always been an important paradigm in software design. By breaking a complex system into pieces, the complexity becomes more manageable. Address spaces provide security by installing firewalls between applications. These two issues are orthogonal, but the distinction has been lost in systems research that has been concentrating on “microkernels” [23].

7 CONCLUSIONS

Building an operating system for emerging computing platforms out of components pays off in terms of flexibility, minimalism, and adaptability. It naturally leads to good software design, rapid implementation, and good portability. Because not all components have to be functional at once, porting and development can be incremental. The TriMedia port was functional in one week, of which five days were spent on learning the new architecture and the development tools. The VLIW architecture of this microprocessor was brand new and different from the ones that MMLite supported at the time. Adding new tools, processors or platforms is easy. Making a device fully functional is still a substantial chore as drivers are often hard to write due to lacking hardware documentation. COM is a good way of adding transparency both to location and privilege level without too much overhead.

The MMLite system is efficient, portable, and has a very small memory footprint that makes it suitable for embedded use. It has successfully been used in several smart I/O cards, sensor boards, and wirelessly connected gadgets. The ability to easily communicate with web services brings the invisible computing platforms into the generic programmable internet cloud.

8 ACKNOWLEDGEMENTS

Thanks to Stefan Sigurdsson who worked on the XML support for MMLite, Adam MacBeth who worked on wireless connectivity, Ray Bittner, Andy Wilson, and Mike Sinclair for building the sensor experimentation hardware.

9 REFERENCES

[1] Miche Baker-Harvey. *ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems*. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, February 1999, New Orleans, USA.

[2] K. Brockshmidt. *Inside OLE, Second ed.* Microsoft Press, Redmond WA, 1995.

[3] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. *Fast Concurrent Dynamic Linking for an Adaptive Operating System.* In the proceedings of the International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis MD, 1996.

[4] Richard Draves, Scott Cutshall. *Unifying the User and Kernel Environments.* Microsoft Research Technical Report MSR-TR-97-10, 16 pages, March 1997. Available from ftp://ftp.research.microsoft.com/pub/tr/tr-97-10.ps.

[5] M. Esler, J. Hightower, T. Anderson, and G. Borriello. *Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project at the University of Washington.* In Mobicom 99, August 1999, Seattle, USA.

[6] Mark Fiuczynski, Richard Martin, Tsutomu Owa, Brian Bershad. *SPINE: A Safe Programmable and Integrated Network Environment.* Eight ACM SIGOPS European Workshop, September 1998, Sintra, Portugal.

[7] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. *The Flux OSKit: A Substrate for Kernel and Language Research.* In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 38-51. ACM SIGOPS, Saint-Malo, France, October 1997.

[8] Johannes Helander, Alessandro Forin. *MMLite: A Highly Componentized System Architecture.* Eight ACM SIGOPS European Workshop, September 1998, Sintra, Portugal.

[9] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. *System architecture directions for network sensors*. ASPLOS 2000.

[10] Integrated Systems Inc. *pSOSystem System Concepts.* Part No. COL0011, May 1995, ISI, Sunnyvale CA.

[11] Michael B. Jones, Joseph S. Barrera, III, Richard P. Draves, Alessandro Forin, Paul J. Leach, Gilad Odinak. *An Overview of the Rialto Real Time Architecture.* In Proceedings of the 7th ACM SIGOPS European Workshop, pages 249-256, September 1996.

[12] Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu. *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities.* In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 198-211. ACM SIGOPS, Saint-Malo, France, October 1997.

[13] Chris Maeda and Brian Bershad. *Protocol Service Decomposition for High-Performance Networking*. In 14th ACM Symposium on Operating System Principles, pages 244-255, 1993.

[14] *CORBA/IIOP 2.2 Specification.* Available from http://www.omg.org/corba/corbiiop.htm.

[15] K. Partridge, L. Arnstein, G. Borriello, and T. Whitted. *Fast Intrabody Signaling.* Demonstration at Wireless and Mobile Computer Systems and Applications, Monterey, CA, December 2000.

[16] M. Rozier, A. Abrassimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser. *CHORUS distributed operating system*. In Computing Systems, pages 305-370, Vol. 1-4, 1988.

[17] Jay Torborg and Jim Kajiya. *Talisman: Commodity Real Time 3d Graphics for the PC.* In Proceedings of SIGGRAPH 96, August 1996.

[18] SAX – Simple API for XML. http://sax.sourceforge.net/

[19] *SOAP Version 1.2*, http://www.w3.org/TR/soap12/, W3C Working Draft, July 2001.

[20] Windows for Smart Cards. http://www.microsoft.com/smartcard

[21] Winsock Direct: fast system area networking for Windows 2000 http://www.microsoft.com/windows2000/en/datacenter/help/wsd_top_overview.htm?id=2129

[22] Zhonghua Yang and Chengzheng Sun. *CORBA for Hard Real Time Applications: Some Critical Issues.* In Operating Systems Review, pages 64-71, Vol. 32-3, 1998.

[23] Michael Wayne Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. Ph.D. Thesis CMU-CS-89-202, Carnegie Mellon University, November 1989.