Java Architectures for Mobilised Enterprise Systems

David Parsons
Massey University
d.p.parsons@massey.ac.nz

Abstract

The coming generation of mobile phones will enable objects from enterprise systems to be distributed across a range of devices of different scales. This object mobility will provide for applications that can take advantage of device-local data and processes to support rich client interaction. However, such mobilised applications bring with them new challenges for the software architect. Distributed objects running on telephones or other small mobile devices will have to work within a number of key constraints, such as limitations on memory and available APIs, and the need to maintain data integrity. This paper focuses on the issues of distributed architectures where objects are able to run on mobile devices but must be universally synchronised. Specifically, it looks at how aspects of current and developing Java APIs can support architectures that synchronise mobilised objects.

1. Introduction

The coming generations of mobile phones (3rd and 4th generation) will enable objects from enterprise systems to be distributed across a range of devices of different scales. Objects from a single application might migrate between application servers, desktop terminals, portable computers, PDAs and mobile phones. Each layer of such a system will provide a different object environment, where data storage technology, run time operating system and available libraries may vary significantly. As objects are distributed across multiple nodes, data integrity must be maintained, even in contexts where connectivity to central data stores is unreliable. Mobile objects will have to be built in such a way that they can easily adjust to their various environments and provide a transparent interface for application developers. For this to be successful, mobilised applications will have to be built to cater for the capabilities of the most constrained devices in the system, which for many applications will be mobile telephones.

For the developer planning to build mobilised systems that include application deployment on 3G telephones, there are a number of available software platforms. Of course each type of device will expose its original equipment manufacturer (OEM) APIs which can be used for development, but since these are proprietary on different devices the same application may have to be re-coded many times. In order to work at a higher level of abstraction and develop software that can work across different types of device, it will usually be more effective to use either the .NET Compact Framework or the Java 2 Micro Edition (J2ME). This paper focuses on the Java platform, which not only provides device interoperability but also supports implementations across multiple operating systems. In the telephone context, windows based systems can only run on the Microsoft Smartphone, which has limited handset support. In contrast, Java can run on all Symbian [1] or Linux operating system devices.

There have been a number of Java based implementations of systems that include mobility, for example Ajents [2] and Klava [3]. However, these have been proprietary modules around lower level Java packages. In this paper we will instead look at how a combination of standard Java technologies (i.e. those ratified by the Java Community Process [4]) might be used to address some of the core requirements of a mobilised computing architecture that includes Java-enabled phones. We begin by exploring some theoretical aspects of mobile computing architectures before reviewing some Java APIs relevant to mobile computing. We then see how these Java APIs might be applied to an implementation of a mobile architecture. We conclude with some discussion about the possible future directions that Java Specification Requests (JSRs) might take in support of this type of architecture.

2. Mobile computing architectures

Mobile computing architectures need to be built with a number of serious constraints in mind. They have to be aware of the limitations of bandwidth, (dis)connectivity, and cope with a disparate range of devices connected to the mobile network. Data needs to be mobilised without compromising integrity. Many of the devices attached to the system will be relatively resource poor, and potentially will be disconnected from the rest of the system either voluntarily (to preserve resources) or in an unpredictable manner (due to loss of wireless connectivity). Any generic architecture must take account of the capabilities of the most limited devices in the system and compensate for their limitations as much as possible.

0-7695-2268-8/05/$20.00 (C) 2005 IEEE

A number of systems have been developed to explore architectures that can meet these various constraints and requirements. Pitoura and Samarso [5] and Jing et al [6] survey a range of these systems, in particular Bayou [7], Odyssey [8] and Rover [9], and explore a number of their features. The most common themes among the mobilised architectures described can be summarised under the general headings of proxy (agent) roles, knowledge representation and transaction management.

Intelligent agents can act as proxies either on the mobile client, the central server or both. These can provide a number of services such as queuing messages or asynchronous remote procedure calls to buffer partially disconnected communications. Client side agents can also be responsible for background pre-fetching into the mobile object cache. In peer to peer systems, agents can be responsible for providing services that require lightweight server implementations on mobile devices. The agent can intercept requests for services and start (and stop) the appropriate service as required. It should be made clear that in this context, although the agents are working within a mobile system, they are not themselves mobile, since mobility is an orthogonal property of agents. Rather, they act as stationary agents, fulfilling the mandatory properties of being reactive, autonomous, goal driven and temporally continuous [10].

Knowledge representation is necessary to enable agents to successfully limit the degree of data transfer without causing excessive cache misses on the mobile device. A degree of intelligence about the mobile objects in the system is therefore required on the part of agents to be able to make sensible decisions about what elements of data should be moved between client and server at a given time. This encompasses aspects such as *differencing* (i.e. only updating the changed parts of mobile objects) *hoarding* (preloading, perhaps by using known queries or keys) and *filtering* (e.g. providing low fidelity data from high fidelity sources, such as using a monochrome image rather than a colour one). In some cases, knowledge about the structure of the object being transmitted can enable customised approaches, such as the caching of directory data described by Cohen et al [11]

The transfer and synchronisation of data in a mobile architecture requires sophisticated transaction management. Transactions in mobilised architectures are by their nature distributed, but can also be long running and can potentially overlap. Since pessimistic transactions are likely to lead to unacceptable performance, some form of optimistic locking is almost certainly a requirement. However, regardless of optimizing policies (e.g. [12]) such locking strategies almost always result in conflicts between concurrent transactions and subsequent transaction rollbacks, or alternatively a system must provide some form of conflict resolution mechanism as an alternative to a 'winner takes all' scenario. In addition to conflict resolution rules, systems can also take advantage of push technology, for example databases can broadcast hotspots where data is frequently updated, and client side agents can use filters to tune to hotpots of interest within a broadcast. Server side agents can potentially work with database servers to target their client agents in a point to point fashion by being aware of the contents of their remote caches. Thus a combination of point-to-point and publish-subscribe push mechanisms can be used to attempt to keep client data in synch with

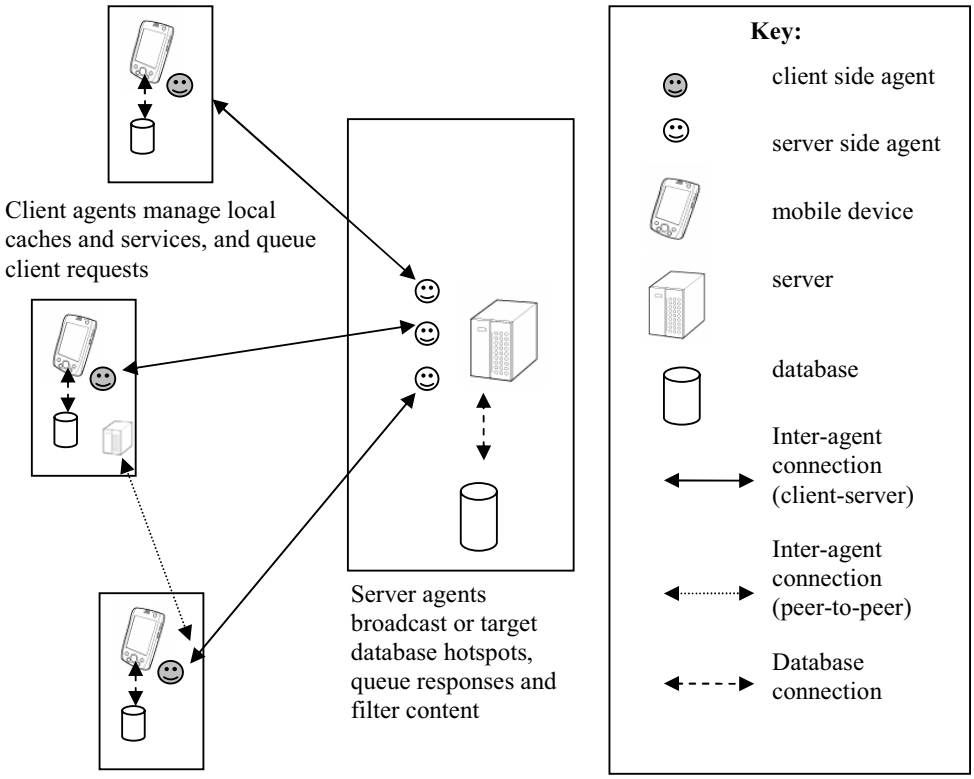

Figure 1. Components of a mobile architecture

server data. Figure 1 summarises the key features of a mobilised architecture that encapsulates many of these approaches.

Many of the systems that have previously been built to explore mobile architectures have been developed using a variety of languages and tools. For example Rover applications are developed using a combination of C, C++ and Tcl and Bayou is essentially based on extensions to Unix. However, any system that hopes to provide interoperable cross platform frameworks needs to be developed using a language and associated tools that support these requirements. The most appropriate current technology to achieve this objective is the Java language, given its 'write once run anywhere' philosophy [13].

3. The Java 2 Platform

Given the various requirements of a mobilised architecture, which spans devices ranging from large servers to small mobile devices, a Java implementation will need to leverage many APIs from across the Java platform. The Java 2 platform is divided into three editions; Java 2 Standard edition (J2SE), Java 2 Enterprise Edition (J2EE) and Java 2 Micro Edition (J2ME). The reason for this separation of editions is that the range of possible Java operating contexts, and the APIs to support them, is so wide that a single set of standard APIs for all types of application and architecture would be excessively large and would include much redundancy. The number of optional packages that spanned all types of system would also be very large. The standard edition is itself divided into core Java (all the fundamental APIs) and desktop Java (the rich client APIs), while the enterprise edition is essentially a superset of the standard edition core. All of the APIs of the standard edition could conceivably be used in enterprise development, and both can use a standard Java Virtual Machine (JVM). The Micro edition, however, comprises various subsets of the core with many additional specialised APIs, and requires customised JVMs.

3.1. The Java 2 Micro Edition (J2ME)

Many mobilised systems (e.g. [14]) are based primarily on the centralized services of large scale application servers using the J2EE APIs, with thin mobile clients using HTTP / WAP connections. However, to fully implement the types of system described in figure 1, it is essential that the mobile devices host rich clients. In this context, it is necessary to run J2ME on the mobile devices to support features such as data persistence, data synchronisation, push technology and messaging. In contrast to the other two Java editions, the micro edition has a special set of requirements because the range of devices that it covers, from tiny embedded systems to pocket computers, is too diverse to enable a single set of APIs to suit all requirements. In addition, the standard JVM is too big for most micro devices, so special JVMs have to be provided for different devices. To cater for this range of devices and requirements, the J2ME architecture provides for a combination of configuration and profile layers that enable particular implementations to be targeted to a given size of device and a suitable API set. In addition, a given device may expose native APIs that can be leveraged by an application specific to that device and, particularly of interest for this paper, optional Java APIs (Figure 2).

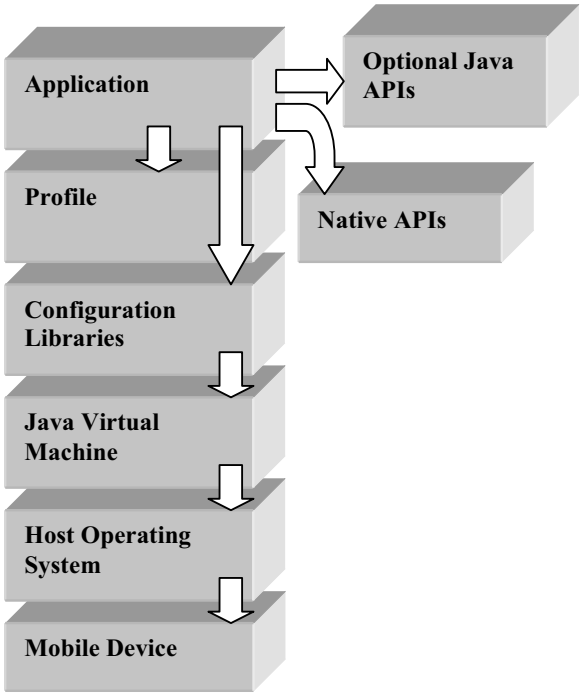

Figure 2. J2ME architecture

In the J2ME architecture, the JVM will be customised for the particular operating system and device, and will support a configuration that defines the minimum set of available JVM features and class libraries for a specific category of devices (e.g. those with very small memories). The profile layer defines the set of application programming interfaces (APIs) available for a particular family of devices (e.g. PDAs, telephones, embedded processors etc.). The only requirement of the target platform is that it must provide a minimal operating system to run the JVM. The optional APIs provide for specialised services to support application specific requirements, such as web services, multi media or data synchronization.

3.2. J2ME configurations and profiles

J2ME configurations specify the minimum requirements for memory, Java language features, VM support and runtime libraries and do not include any optional components. There are two defined

configurations, the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). These are published as standard specifications as part of the Java Community Process (JCP). PDA style mobile devices will typically support the CDC, whereas for mobile telephones, CLDC is the relevant configuration

The CLDC requires a very small virtual machine that does not have to comply with the full JVM specification, nor does it have to support the complete Java language specification. API support is limited to a selection of classes from the java.lang, java.lang.ref, java.io and java.util packages, along with the Generic Connection Framework (GCF) classes in the javax.io.microedition package. These limitations can, of course, impact on how mobile architectures can be implemented.

J2ME profiles are supported by a specific configuration, and are used to provide functionality specific to a family of devices, vertical market or industry. Applications may have access to more than one profile API at runtime so that they can be combined in appropriate ways to provide an API set that is geared towards a specific application requirement.

The key profiles for mobile phone development are the Mobile Information Device Profile (MIDP) which provides for interactive applications with graphical interfaces and persistent data, and the Wireless Messaging API (WMA) which supports short messaging. The MIDP and WMA profiles are core components of the Java Technology for the Wireless Industry (JTWI) specification, which aims to provide a standard API set for the mobile phone market. JTWI also includes a conditional (optional) component, the Mobile Media API (MMAPI). In addition, there are further optional J2ME APIs that are either currently available or being developed as part of the JCP. These include, among many others, the Location API, J2ME Web Services and the Data Synch API. These optional APIs can be installed on mobile devices in whatever combination is required for a given application, providing, of course, that the actual implementations of the specifications can be provided with a small enough footprint to work within the constraints of a given device.

Many of the most challenging features of a mobilised architecture relate to the distribution of persistent data and its synchronisation. To address these issues, we need to build an appropriate persistence layer that can be applied across all nodes of a mobilised system and integrate into it an appropriate synchronization mechanism. In this paper we suggest that the Java 2 platform can provide us with the tools to address both of these requirements, using Java Data Objects (JDO) and Java Data Synch respectively, though further work needs to be done on enabling these specifications across mobile architectures and integrating them with J2ME systems. In particular, JDO implementations will need to be tailored for small devices to enable a mobile, distributed object model.

4. Java persistence and JDO

Java provides a number of standardised options for object persistence, but not all of these can be used in all three editions of the Java 2 platform. For enterprise level systems, a separate database management system is a requirement, which can be mapped to an object model via various Java APIs. J2SE supports object persistence through serialization (to flat files or into database columns) and JDBC (for mapping to relational databases), while the enterprise edition adds entity Enterprise Java Beans (EJB) as a persistence mechanism. The problem for mobile systems is that some configurations of J2ME do not automatically support any of these persistence mechanisms, since the CLDC includes only a very limited set of classes from the java.io package (no file or object readers or writers). As an alternative set of APIs, MIDP provides its own set of APIs (the Record Management System), analogous to flat file storage, in the javax.microedition.rms package. The RMS comprises variable length record stores, each of which is a collection of variable size binary data records. Each record store has a unique name, and each record within a store has a non-reusable integer index that acts as a primary key. A record store is only accessible to the MIDlet (MIDP application) that created it. Although individual operations on a record store are atomic, there is no transactional support apart from a version number that can be used to support manually implemented locking strategies. This mismatch between persistence APIs on mobile clients and servers can be problematic if we are trying to distribute a persistent object model across all types of node in a mobile architecture. Fortunately, Java provides another option for object persistence, the Java Data Objects (JDO) specification. This is an optional component of the standard edition that can also be used in enterprise development via the Java 2 Connector Architecture. Importantly, JDO can also be used in a mobile environment, and in fact it is a recommended persistence technology for mobile devices [15]. JDO can be a very useful tool to encapsulate a distributed object model, because we can use its APIs to render transparent the different underlying data stores being used on different nodes, and encapsulate services such as data synchronization. At present, there are no implementations of JDO for JTWI devices but there are several that will work on PDAs. JDO implementations based on MIDP RMS have been explored (e.g. [16]) and will be available at some point, though due to the limitations of MIDP they may be only API based and not include runtime metadata.

5. Java data synchronization

If wireless networks were free to use, had infinite bandwidth, had very high quality of service and universal coverage then mobile devices would be able to maintain data integrity as easily as desktop clients of remote databases. However, due to the limitations of the infrastructure that exists now, and is likely to exist for a

long time into the future, there will be a need to synchronize data between mobile clients and database servers after a period of disconnected operation. Synchronization is a key enabler for interactive networked applications, since most enterprise applications are data centric.

There are many propriety products that provide data synchronisation, and there have also been some concerted efforts to standardise these protocols, such as the OMA (Open Mobile Alliance) data synchronization working group and the SynchML initiative, which joined forces in 2002. SynchML was founded by Ericsson, IBM, Lotus, Motorola, Nokia, Palm, Psion and Starfish to release an open data synchronization protocol to ease data exchange across networks, platforms and device types [17]. SynchML has been adopted by a wide range of vendors. For example Pointbase, one of the main vendors of mobile databases, is SynchML compliant. SynchML is XML based, supports a range of data transport protocols such as WSP (WAP Wireless Session Protocol), HTTP, OBEX (Bluetooth, infra-red), and addresses the resource limitations of mobile devices. This means that the protocol is designed to fit within the memory footprint of devices such as mobile phones. To support this ability, data can be exchanged using a binary format (WBXML) to reduce both the memory required and the load on processor resources.

From the Java perspective, the key synchronization component is Java Specification Request (JSR) 230, the Java Data Synch API. This API is intended to enable mobile applications to synchronize their local application specific data with corresponding server side data, replicating any changes made to the data on either client or server. It is built primarily on existing industry support for a universal standard for data synchronization, so it is unsurprising to find that many members of the expert group (including the mobile phone manufacturers Motorola and Nokia) are also members of OMA/SynchML. Like all Java APIs, the intention of the Java Data Synch API is to provide a high level API though a generic interface to the whatever data synchronization implementations are actually being used by the system devices. The specification provides Java interfaces while vendors will provide implementation classes to synchronize data via underlying protocols (either native or Java based). SyncML Data Synchronization is one of these underlying protocols, and the specification will be based on this and other existing mobile data synchronization frameworks.

The Data Synch API is planned as an optional package within J2ME that could be used with either CLDC or CDC. This means that it could be used with MIDP on small mobile devices. The expert group was formed in October 2003, and there are currently 16 members, with the specification request being led by a representative of Siemens AG. It is expected that the specification will be adopted by the Java Community Process by early 2005. With a Java Data Synch API in place, a mobile JDO implementation could encapsulate data synchronisation behind a mobile object model.

6. Integrating broadcast and point to point messaging

To support the distribution and synchronization of data across multiple devices, support for messaging triggers is required to signal when data updates are due. Both broadcast and point to point messaging can be implemented using push technology, which provides the ability to push information to a device without that information being specifically requested, enabling live, transparent updates to applications and data. A fairly basic example of this is already provided with WAP push technology. The more sophisticated push support in MIDP enables us to send a message to a running application, trigger an application update, provide user alerts, send data, notify listeners or start an application using the Application Management System (AMS). The original MIDP specification (version 1.0) was lacking in support for push services, providing only HTTP based connectivity, which is a client-server request-response mechanism. Using HTTP, it is not possible to push data since the initial message must always be a request to the server, i.e. it must be client initiated. Bonnet at al [18] noted that this limitation precluded any peer to peer communication in their Message Board Client system. Similarly, Roth and Unger [19] report how the lack of such facilities, along with limited server socket support in an early version of the Java-integrated Waba platform, led to excessive dependency on client initiated communication in their QuickStep system.

To provide for a richer set of connectivity options and overcome these types of issues, push support was included in the MIDP version 2.0 specification. Push connections in MIDP may be implemented using Transmission Control Protocol (TCP) sockets or User Datagram Protocol (UDP) datagrams [20]. Such technologies allow servers to push data to clients, and also enable peer to peer two-way Java based communications. The various architectural components are shown in Figure 3.

Due to some optional features of the specification, it is difficult to guarantee which particular connection types might be supported by a given mobile phone. An alternative way of supporting push is to use the Java Wireless Messaging API (WMA) component of JTWI, which includes a Short Message Service (SMS) point to point and Cell Broadcast SMS (CBS) publish subscribe push option. Since SMS is now a standard feature of mobile phones, this type of connection is virtually guaranteed to be available on a 3G phone.

Regardless of the underlying communication mechanism being used, the MIDlet handles pushed connections using the push registry. Figure 4 shows the main features of the MIDP push registry, comprising a simple API and two types of push events; inbound connections and timer alarms. Timer alarms enable an

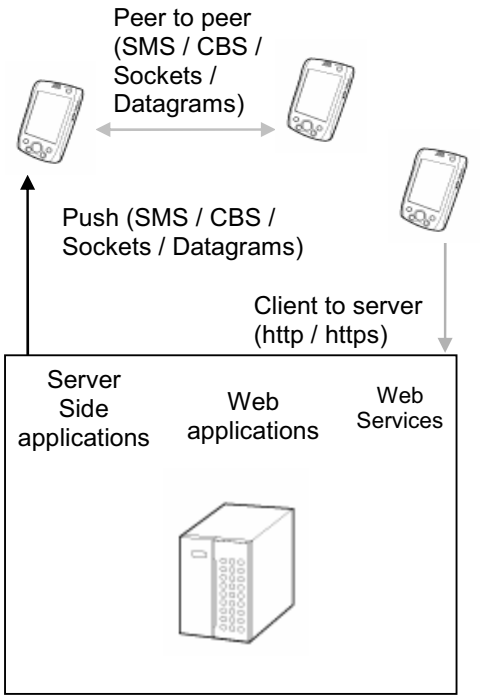

Figure 3. Components of a mobile application architecture

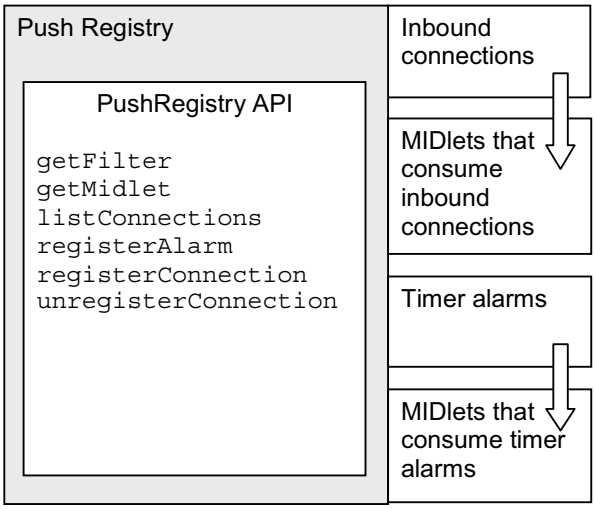

Figure 4. The MIDP push registry

application to be started or updated on a timed basis. This does not require any external connection, so the source of the push event comes from the device itself. In contrast, inbound connections enable an application to be started or updated from an external source, which may be a peer (i.e. another mobile device) or a server side application. Since the wireless messaging API is slated to be available as part of J2SE via the Generic Connection Framework optional package [21], and the Server API for Mobile Services is also likely to be available soon, desktop or server based systems can be developed to provide the control framework for push-enabled application services.

In our mobilised architecture, timer alarms could be used to monitor the mobile cache for hoarding, while inbound connections could be used to trigger updates broadcast from servers or sent point to point by server side agents.

7. Applying a Java services stack to a mobile architecture

In this paper we have reviewed both the components of a mobilised architecture and a number of Java APIs. In this section we will draw the two together to see where Java can be applied to a mobile architecture.

From Figure 1, we can see that the key components of our architecture are the agents, with the database components being a core feature. Assuming that the mobile devices in the system are Java enabled, the client side agents can make use of the MIDP APIs, in particular the push registry, to meet their requirements. The push registry can be used to trigger agent activity in managing the mobile cache. The agents will need to be integrated with the persistence layer to make this work effectively, so it is likely that the JDO persistence layer will encapsulate some agent activity. The persistence agents will be able to utilise the Java Data Synch APIs to manage the cache. The push registry can also be used to trigger request for mobile server connections by peer devices. Once a need has been identified for data download the agents can use HTTP connections to use services from J2EE APIs such as servlets, JSPs or web services

On the server side, agents can again use the JDO and Java Data Synch APIs to manage distributed data. In addition, server side push can be implemented using the Wireless Messaging APIs, as specified for the standard edition of Java, and of course the general features of the Java 2 Enterprise Edition. For example, server side agents could be plugged into the servlet filter layer of an enterprise system to pre-process HTTP client-server interaction. The Java services stack is outlined in Figure 5.

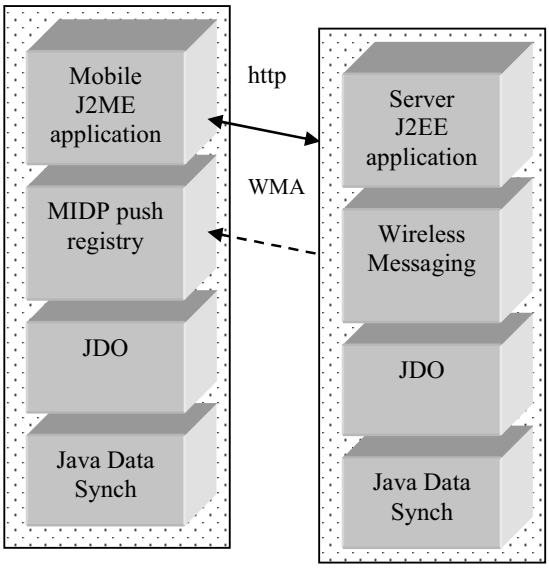

Figure 5. Java services stack for mobile applications

8. Java Specification requests and future mobile architectures.

In this paper we have identified a small number of core Java specifications that are required to build a mobilised software architecture using standard Java APIs. Some of these (e.g. Java Technology for the Wireless Industry and J2EE) already provide viable standards and implementations for such architectures. In contrast, although it is fully specified, Java Data Objects does not yet provide us with an implementation that can run on the smallest mobile devices (i.e., those that use the MIDP APIs), but we may expect these to be developed. Some other relevant specifications are still at the request stage within the JCP and have not yet been published. Most importantly for this discussion, the Java Data Synch API is currently in this state. Of the nearly 300 specifications requests listed by the JCP [22], many are in a state of flux, and not all will necessarily be accepted, or may be withdrawn. However, JSR such as JSR 87 (Java Agent Services) and JSRs 232 and 233 (Mobile Operational Management and J2EE Mobile Device Management and Monitoring Specification) may be of real value in the future.

9. Related work

The architectural framework described here is based on pooled features from a number of systems surveyed by [5] and [6], and the combined features of a number of current and proposed Java APIs. However there are some alternative architectures, variations and related technologies that might be considered. Burge et al [23] propose a system based on pervasive store-and-forward data stores that gather data from roaming devices via Jini [24]. Unlike the architecture described here, responsibility for ensuring that data is reconciled with the central database lies with the mobile storage rather than the mobile device. Wolfe [25] also indicates that potential ubiquity of Jini enabled devices could support such architectures, despite the initial slow take up of the technology.

Lunney and McCaughey's [26] discussion of remote persistence implementations with Java focuses on socket level and RMI connections, describing an applet/RMI client architecture where persistence is confined to the server. However, their architecture could be extended to include client side object caches. RMI based systems in particular do have the advantage of serializing objects between client and server, whereas the Java Synch model described here will probably use an intermediate data format (if it follows the lead of SyncML.) Also, although their discussion is limited to applets as the RMI client, J2ME can support a specialised RMI layer [27], though the current specification is unavailable on the smaller mobile devices that use CLDC.

The Java APIs for Integrated Networks (JAIN) are another feature of Java in the telecommunications field that could be considered in a mobilised enterprise system, though they do not impact specifically on the generic architecture described in this paper. For example, Tsuei and Sung [28] describe a 'JAIN-like platform' for ubiquitous information services, including WAP based mobile Java systems. However, they do not address the JAIN APIs directly. The intention of JAIN is to provide standard Java APIs for converged IP and PSTN networks, though there have been some problems finding support for some aspects of the service provider APIs, and seven specification requests in this area have been withdrawn [29].

One of the more common features of experimental mobilised systems is the use of mobile, rather than static, agents. In this paper we describe a system of client and server side agents that manage the transfer of data, but they themselves remain on their host devices. In contrast, many systems have been described where the ability of agents to migrate between, and execute on, multiple devices is exploited. Java is the language of choice for many agent based system, since the mobile agents take advantage of dynamic loading to transfer programs, data and metadata between devices [30]. Agent based systems can enable complex interactions between agents and resources so that disparate programs and data can be marshalled to achieve a specific task An example of this type of systems is StratOSphere [31], though it should be noted that this is not necessarily targeted at small mobile devices, rather, the focus is the mobility of code. However, there may be some benefit to the integration of mobile agents into the architecture described here. The main limitation in terms of building a standardised framework from Java components might be the slow progress of standardisation. Whereas a static agent can easily be encapsulated behind another local API, such as

a Java Data Objects implementation, a mobile agent system would require exposing more proprietary APIs.

Mobile agents rely on the dynamic loading capabilities of their host virtual machines, and a similar technique of dynamic loading is used in the MOCA system, though rather than being based on autonomous agents, the focus here is on a service oriented architecture [32]. The technique used is similar in principle to a combination of Jini style features and Java embedded server. Again, aspects of a service oriented architecture might complement the approach described in this paper, and could be regarded as being orthogonal to it rather than providing a completely different approach.

10. Conclusion

In this paper we have described a generic architecture for mobilised systems drawn from a number of previous examples, and explored the relationship between the various components of this architecture and a number of Java specifications, either current or in progress. In particular we have focused on those APIs that enable the effective management of distributed data and two way communications between clients, servers and peers in systems that include the smallest of mobile devices. Once a critical mass of the relevant specifications has been fully published and implemented, we will have the opportunity to move from experimental and proprietary architectures for mobilised enterprise systems to standardised, platform independent, interoperable frameworks. This should foster more reuse and reliability and enable the building of more efficient and powerful mobilised systems. Most of the components of the necessary J2ME architecture are in place. Critical work, however, has to be done in the areas of integrating JTWI compatible JDO implementations with the Data Synch API if such frameworks are to transparently manage the mobilisation of data.

References

[1] Jode, M.D., *Symbian on Java*, Symbian, 2004, http://www.symbian.com/technology/SymbianOnJava2_05.pdf, last accessed June 17th, 2004

[2] Izatt, M., P. Chan, and T. Brecht, "Ajents: towards an environment for parallel, distributed and mobile Java applications", *Software: Practice and Experience*, 2000. **12**(8): p.667-685.

[3] Bettini, L., R.D. Nicola, and R. Pugliese, "KLAVA: a Java package for distributed and mobile applications", *Software: Practice and Experience*, 2002. **32**(14): p.1365–1394.

[4] SunMicrosystems, *Introducing the Java Community Process^{SM} (JCP^{SM}) Program Version 2.6*, Java Community Process, 2004, http://jcp.org/files/whitepaper.JCP26Whitepaper.pdf, last accessed June 24th, 2004

[5] Pitoura, E. and G. Samaras, *Data management for mobile computing*. 1998, Boston: Kluwer.

[6] Jing, J., A. Helal, and A. Elmagarmid, "Client-Server Computing in Mobile Environments", *ACM Computing Surveys*, 1999. **31**(2): p.117-157.

[7] Demers, A., K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. "The Bayou Architecture: support for data sharing among mobile users". in *Workshop on Mobile Computing Systems and Applications*. 1994: IEEE. p.2-7

[8] Satyanarayanan, M., B. Noble, P. Kumar, and M. Price. "Application-aware adaptation for mobile computing". in *ACM SIGOPS European Workshop*. 1994. Wadern, Germany: ACM Press. p.104

[9] Joseph, A., J. Tauber, and F. Kaashoek, "Mobile computing with the Rover toolkit", *IEEE Transactions on Computers*, 1997. **46**(3): p.337-352.

[10] Lange, D. "Mobile Objects and Mobile Agents: The Future of Distributed Computing?" in *ECOOP 98 - Object Oriented Programming*. 1998. Brussels, Belgium: Springer-Verlag. p.1-12

[11] Cohen, D., M. Herscovici, Y. Petruschka, Y. Maarek, A. Soffer, and D. Newbold. "Personalized Pocket Directories for Mobile Devices". in *Proceedings of the eleventh international conference on World Wide Web*. 2002. Hawaii, USA: ACM Press. p.627-638

[12] Perich, F., A. Joshi, Y. Yesha, and T. Finin. "Neighborhood-consistent transaction management for pervasive computing environments". in *Database and Expert Systems Applications*. 2003: Springer-Verlag. p.276-286

[13] Gosling, J., B. Joy, and G. Steele, *The Java Language Specification*. The Java Series. 1996, Reading, Mass.: Addison-Wesley.

[14] Zeadally, S. and J. Pan, "J2EE support for wireless services", *Journal of Systems and Software*, 2004.

[15] Reese, G., *Java^{TM} Database Best Practices*. First ed. 2003, Cambridge: O'Reilly.

[16] Braig, A. and S. Gemkow, *The BonSai Principle - Persistenz in der Java 2 Micro Edition*, in *Java Spektrum*. 2002.

[17] Heintzman, D., *SyncML*, SyncML Initiative, 2002, http://www.openmobilealliance.org/tech/affiliates/syncml/introducing_syncml_29jan02_douglas_heintzman.pdf, last accessed June 17th, 2004

[18] Bennett, J., M. Armstrong, and S. Gupta. "A Message Board Client for handheld devices". in *42nd ACM annual Southeast regional conference*. 2004. Huntsville, Alabama: ACM Press. p.17-18

[19] Roth, J. and C. Unger, "Using Handheld Devices in Synchronous Collaborative Scenarios", *Personal and Ubiquitous Computing*, 2001. **5**(4): p.243-252.

[20] Ortez, E., *The MIDP 2.0 Push Registry*, Sun Microsystems, 2003, http://developers.sun.com/techtopics/mobility/midp/articles/pushreg/, last accessed June 23rd, 2004

[21] Ortez, E., *The Wireless Messaging API*, Sun Microsystems, 2002, http://developers.sun.com/techtopics/mobility/midp/articles/wma/, last accessed June 23rd, 2002

[22] JCP, *List of all JSRs*, Java Community Process, 2004, http://www.jcp.org/en/jsr/all, last accessed June 25th, 2004

[23] Burge, L., S. Baajun, and M. Garuba. "A Ubiquitous Stable Storage for Mobile Computing Devices". in *2001 ACM symposium on Applied computing*. 2001. Las Vegas, Nevada, United States. p.401 - 404

[24] SunMicrosystems, *Jini™ Architecture Specification*, Sun Microsystems, 2001, http://wwws.sun.com/software/jini/specs/jini1_2.pdf, last accessed September 13th, 2004

[25] Wolfe, A., *Java is Jumpin', This Time for Real*, in *ACM Queue*. 2004. p.16-19.

[26] Lunney, T. and A. McCaughey. "Object persistence in Java". in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*. 2003. Kilkenny, Ireland. p.115-120

[27] Hodapp, M., *JSR 66: J2ME RMI Optional Package Specification Version 1.0*, Sun Microsystems, 2002, http://www.jcp.org/en/jsr/detail?id=66, last accessed September 13th, 2004

[28] Tsuei, T. and C. Sung, "Ubiquitous information services with JAIN platform", *Mobile Networks and Applications*, 2003. **8**(6).

[29] Sun, *JAIN General Q&A*, Sun Microsystems, 2004, http://java.sun.com/products/jain/qa.html, last accessed 2004

[30] Wong, D., N. Paciorek, and D. Moore, "Java-based Mobile Agents", *Communications of the ACM*, 1999. **42**(3): p.92-102.

[31] Wu, D., D. Agrawal, and A.E. Abbadi. "StratOSphere: Mobile Processing of Distributed Objects in Java". in *4th annual ACM/IEEE international conference on Mobile computing and networking*. 1998. Dallas, Texas, United States: ACM Press. p.121-132

[32] Beck, J., A. Gefflaut, and N. Islam. "MOCA: A Service Framework for Mobile Computing Devices". in *1st ACM international workshop on Data engineering for wireless and mobile access*. 1999. Seattle, Washington, United States: ACM Press. p.62-68