MobiMan: Bringing Scripted Agents to Wireless Terminal Management[*]

Venu Vasudevan, Sandeep Adwankar, and Nitya Narasimhan

Mobile Platforms and Services Department, Motorola Labs,
1301, E. Algonquin Road, ILO2-2240, Schaumburg, IL 60196
{venuv,adwankar,nitya}@labs.mot.com

Abstract. The increasing software complexity of wireless devices and wireless data service provisioning motivates a wireless terminal management challenge. The systems management solution for this problem needs to scale up to large device populations, while being lightweight enough to be pragmatic for resource-constrained devices. The work in this paper builds upon the emerging *SyncML* standard for wireless terminal management in order to bring sophisticated policy-based management to large populations of wireless data devices. It is anticipated that this technology will simplify the upgrade and management of wireless data devices substantially, thus encouraging the adoption of sophisticated data terminals.

1 Introduction

The increasing complexity of wireless devices and services motivates an automated terminal management challenge. Complex client-side wireless tools (e.g. WAP browsers) need to be remotely configured upon service activation or upgrade. Mobile service operators desire the ability to dynamically upgrade applications and services on a mobile device, motivating the need for scalable software distribution capabilities. Effectively supporting a complex palette of applications on a consumer-oriented device requires pro-active diagnosis and troubleshooting of both the devices and the network. While these tasks can be done in an operator-assisted fashion, the absence of a *scalable, automated management infrastructure* can contribute to a high total cost of ownership. Early studies [KD99] estimate this number as being several times the retail cost of the device

The goals of wireless terminal management resemble those of "classical" (wired) systems management in terms of scaling and automation. However, achieving the goals in a resource-constrained, intermittently connected environment presents unique technical and business challenges. The pervasiveness of the Simple Network Man-

[*] We would also like to thank Kevin Cutts and Hung Tsang from Motorola's PCS business unit for many constructive discussions in this context.

agement Protocol (SNMP) and its thick-manager, thin client-agent architecture make it a logical first choice. However, while SNMP agents can be deployed on thin devices, traditional client-server SNMP lacks the "elastic server" or delegation model [GY95] that is required for effective management in intermittently connected environments.

Advanced proposals within the SNMP community (such as the SNMP mid-level manager architecture [SMLM, SNMX]) provided a standards-compliant basis for a delegation model but failed to gain the widespread traction of core SNMP. Mobile agent technology can provide a delegation model for systems management that is agnostic to the management protocol. However, mobile agent infrastructure is too heavyweight for mobile devices, requiring some programming features (e.g. dynamic classloading) that are not supported by current mobile device application platforms like J2ME.

From a pragmatic point of view, another problem with SNMP is its lack of uptake in the mobile wireless space. The wireless industry, which seeks a single, integrated standard to manage device-resident information as well as device operation, has rallied around SyncML [SM02, JN01] and SyncML for Device Management (SyncML-DM)[1] [SD02] as the management standard of choice. SyncML caters to both the information management and device management needs of the wireless industry, while supporting a lightweight architecture using an XML-based command language.

While SyncML-DM is best suited for simple request-response management operations (analogous to a single SNMP *Get* and *Set*), complex operator management functions require richer predicates and procedures to capture the necessary semantics. For instance, a network operator may want to run an automated management test on all cell phones in the "847" area code. Furthermore, he may want this test to run during off-peak hours and only on terminals with sufficient battery-power levels. Although basic SyncML-DM provides the low-level mechanisms to collect the relevant telemetry, expressing such activation policies as a collection of primitive SyncML-DM interactions could be prohibitively expensive.

1.1 The MobiMan Architecture

In the **MobiMan** architecture, we explore an alternative "embrace and extend" approach to bringing the benefits of *scripted mobile agents* to SyncML-based wireless terminal management. MobiMan defines a SyncML-derived scripting language called ***Symple*** (**SY**nc**M**L **P**rogramming **L**anguag**E**) that extends SyncML semantics in a lightweight manner suitable for resource-constrained devices. The MobiMan architecture also extends the SyncML runtime framework with support for the scheduling, evaluation and lifecycle management of Symple agents. To strike a balance between capabilities and deployment costs, we designed Symple according to the following principles:

- *Embeddable within SyncML.* Symple agents can be embedded within standard SyncML packets, allowing us to reuse SyncML as an agent distribution protocol.

[1] Henceforth, the terms SyncML and SyncML-DM will be used interchangeably to mean SyncML-DM.

- *Lightweight.* Symple is easy-to-learn, requiring only a small amount of code to define complex requirements that can be interpreted by a lightweight client-side SyncML platform.
- *Extensible.* Devices can support different variants of Symple in accordance with their resources and computing capabilities, with Synclets being discarded without error by Symple-unaware, SyncML capable devices.

Given that the SyncML standard is of recent vintage, we provide a quick tour of SyncML in Section 2, followed by an overview of the MobiMan architecture in Section 3, with focus on the computing elements (Synclets) and the client-side runtime "container" architecture (Micropods). Section 4 delves deeper into the structure of Synclets, and their support for conditional execution. We conclude the paper with a discussion of our experiences building a Synclet-based system on Motorola wireless devices, future directions for our research, and comparisons of our work to related ideas in distributed systems management.

2 SyncML: A Short Tour

The SyncML standard[2] [SM02] was developed as XML-based information synchronization standard designed specifically for the needs of the wireless industry. Thus, the SyncML protocol is lightweight to cater to device limitations, language-neutral and protocol-neutral. To suit device limitations, SyncML language constructs are kept fairly minimal, with the protocol not using some features (e.g., server sockets) that are yet to become pervasive on mobile devices. Because SyncML is XML-based, it is inherently language-neutral, and supports OEM-specific extensibility. In addition, the SyncML "protocol" can run over a number of underlying transport protocols including HTTP, WSP, and OBEX.

SyncML's popularity in information synchronization led to its scope being expanded via SyncML-DM [SD02] to include device management. SyncML-DM allows management actions to be performed on management objects, where a management object might represent a device configuration or the run-time software application environment. Actions taken against the former might include reading and setting parameter keys and values, while actions taken against the latter might include installing, upgrading, or uninstalling software elements. The signatures of the Get and Set methods on a management object are type-specific and may vary substantially in complexity. For instance, a management action for setting the device clock accepts a simple textual MIME type (text/plain), while an action to change the WAP browser settings requires new WAP provisioning "blobs" to be transmitted as part of the Set operation. Software upgrades present another example of a management action with significant payload complexity.

The SyncML-DM is a 2-phase protocol consisting of a *setup* phase for authentication and device information exchange, following by a *management* phase that can be

[2] It was developed first in a separate standards body, which has since been merged with the Open Mobile Alliance (OMA).

repeated multiple times to support complex manager-to-mobile sessions. A management session may also start with Packet 0 (the trigger), where the trigger may be out-of-band depending on the environment

While SyncML-DM significantly expands the scope and utility of SyncML to include device management, it is limited in the following ways :

1. SyncML-DM based device management is limited to *terminal-at-a-time* management. It allows a cellular operator to run only one diagnostic operation at a time on each terminal.
2. SyncML-DM does not support intelligent postponement of management operations (e.g. postpone terminal operation until battery level is above 50%), something which is necessary to scale terminal management to large terminal populations.
3. SyncML-DM does not allow operators to schedule *coordinated* terminal management operations across collections of terminals.

These limitations restrict the subset of "Opex" (operational expense) minimizing management functions that a cellular operator can perform using SyncML. The goal of MobiMan is to add a more powerful computing abstraction to SyncML-DM to facilitate more comprehensive, automated, scalable systems management

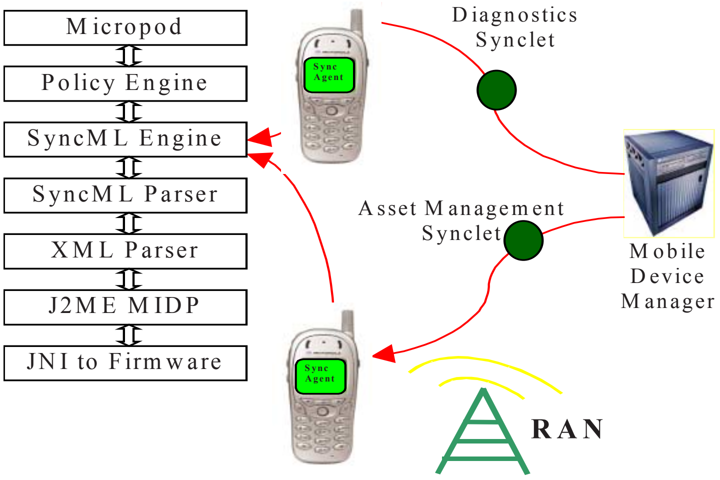

Fig. 1. The MobiMan runtime architecture

3 MobiMan: Architecture and Runtime

As shown in Figure 1, MobiMan augments the standard SyncML-DM runtime framework in terms of both the basic execution entities and the execution model. In doing so, it extends the SyncML-DM framework with two elements: ***Synclets*** and ***Micro-***

pods. ***Synclets*** are scripted agents written in Symple, allowing complex sequences of management instructions to be expressed in a single executable object. ***Micropods*** are terminal-resident containers that have the ability to receive Synclets and manage their lifecycle; for instance, micropods can activate and deactivate Synclets based on state.

The key to the MobiMan container model is to organically expand SyncML to support *intelligently postponable* computing objects, whose execution triggers are sophisticated. A cellular operator can exploit this sophistication to perform large management operations in a manner that conserves bandwidth, avoids terminal operations when the terminal is in an unsuitable state, and coordinates complex multi-terminal operations with complex orchestration policies. We describe Synclets and Micropods in detail in the following sections.

3.1 Synclets and Synclet Bundles

Synclets are the basic unit of computation in *MobiMan*. A Synclet is an executable script consisting of Symple commends, where Symple is an extension of SyncML. A Synclet specification comprises of two parts: a *policy* and an *action routine*. The Synclet *policy* specifies non-functional aspects of the Synclet such as if, when and how often it should be executed. The *action routine* is the functional code that makes up the Synclet. Synclets are interpreted and executed by the extended client-side SyncEngine known as a Micropod.

Synclets are transported over the SyncML protocol, which in turn is bearer-agnostic. Thus Synclets may be transported over HTTP, SMTP or other IP-based protocols. To be compatible with Synclet unaware clients, Synclets are carried as the payload – i.e., as nested tags – in a *Synclet* XML element within the SyncML body. Standard SyncML engines that are not MobiMan-enabled will simply ignore the Synclet scripts within the enclosing Synclet tags. As Synclet upload and execution are decoupled, multiple Synclets with differing execution policies may be carried in a single SyncML session, i.e. nested within the same *MobiMan* tag.

Synclet *bundles* are a convenience mechanism, analogous to Java packages or modules in programming languages. The bundle mechanism allows a group of Synclets to be loaded in a single SyncML session, and be henceforth accessed by the remote operator using a single bundle name. The bundle notion facilitates packaging, distribution and policy management of Synclets.

3.2 Micropods

Micropods are terminal resident containers layered over the standard client-side *SyncEngine*. They host and manage Synclets that are dispatched to the terminal. Micropods perform functions relating to Synclet lifecycle management, and serve as secure sandboxes for Synclet execution. Lifecycle functions include (de) activation, multitasking between concurrently executing Synclets, and the suspension and the revival of Synclets that were stopped due to adverse terminal conditions (e.g. deteriorating battery level). To avoid race conditions on the device, Micropods support "virtual multitasking" where multiple terminal-resident Synclets might have partially executed at any point in time, but only one Synclet is actually executing at any instant.

As previously mentioned, Synclet specifications include a model of conditional, policy-based execution analogous to the UNIX *cron* model. Micropods augment the standard SyncEngine with a *policy engine*, which determines the subset of Activatable Synclets based on evaluating individual Synclet policies. Policies could be based on absolute time, invocation cardinality (number of times a Synclet should be run), and device state. More sophisticated micropods could support the interleaved execution of multiple synclets. These could include allowing a certain maximum number of concurrent synclets, deadlock avoidance or resolution between concurrent synclets, and fairness policies for Synclet swap-out whereby idle synclets are swapped out for other waiting synclets.

As with Java applet environment and J2EE servlet containers, micropods provide a constrained environment to executing synclets, bounding and monitoring their access to the underlying terminal. Synclets are privileged applications have limited access to retrieving and modifying the device state via a special set of "closed classes" in Java. Two factors simplify sandboxing in MobiMan. First, the Synclet dispatching capability is accessible only to a small number of trusted parties (namely service operators). So authentication solutions such as digital signatures can check the signature against a very small universe of trusted sources. Secondly, the use of a scripting language allows for language safety verifiers to be developed, and for sandboxes to suspend (and resume) the script at arbitrary points in the program.

Synclets face the unique situation in executing on mobile wireless devices, namely that the device may be powered off at any time without operator control. Micropod lifecycle management techniques need to allow for script re-entrancy in such adverse situations. Script re-entrancy might involve re-prioritizing Synclets when they are restarted, to reflect the new environment (analogous to adjusting the *nice* value of processes in Unix). This is handled by re-evaluating Synclet policies of awakening Synclets in the changed environment.

4 More Synclet Anatomy

As described previously, Synclets consist of *policy* and *action routine* components. The action routine is the body of what the Synclet does, and the policy is the trigger condition for the Synclet. The policy, action routine separation allows the same functional Synclet to be reused in different circumstances. This section elaborates on the kinds of policies supported in Symple, and the SyncML extensions supported in an action routine. SyncML is extended in two ways in the action routine language: the addition of a few new commands, and support for conditional command execution (aka *guarded* commands)**.** environment.

4.1 Policies

Policies are specifiable at three levels of granularity: *terminal*, *synclet bundle*, and *synclet*. *Terminal* level policies apply to all scripts that will execute on the terminal from the time the policy is installed. For instance, a terminal policy may dictate that only one Synclet shall be active at a time. *Synclet bundle* policies apply to a collection of synclets that were loaded as an aggregate, perhaps because they collectively perform a single cohesive task. *Synclet* policies govern the execution of the Synclet's action routine, and may include the maximum time a Synclet is allowed to run, resources that should be allocated before the Synclet should be activated, or recovery actions when Synclet execution is interrupted. Terminal policies tend to be more static and persistent than Synclet bundle and Synclet level policies.

4.2 Extended SyncML Command Set

SyncML provides a fairly minimal set of commands for device management, but is missing some important primitives that are building blocks for device and service management. *Symple* currently extends the SyncML command set with three commands: *assert*, *schedule*, and *perform*. All these commands may contain a redirect block (see section 4.3 below).

```
<Synclet>
  <Guard>
     <Attribute> Battery_level </Attribute>
     <Condition> GREATER_THAN </Condition>
     <Threshold> 5 </Threshold>
  </Guard>
  <Assert>
     <Item>
        <Target>
           <LocURI>
              ./Sync/DM/WAP/WAPSTNG2/GRPS_APN
           </LocURI>
        </Target>
        <Data>
           internet2.voicestream.com
        </Data>
     </Item>
  </Assert>
</Synclet>
```

Fig. 2. Synclets with guarded commands

1. *Assert* (see Figure 2) allows the Synclet to assert a certain device (or network) condition, and take an exception action in case this isn't true. Assert is useful where the absence of a condition (e.g. non-null values for WAP session parameters) requires corrective action.
2. *Schedule* is used to represent timed and/or repetitive commands. Parameters in the schedule command may specify the maximum number of times the command is executed, the time interval between successive iterations, and the delay between the loading of a Synclet and the first "run" of the scheduled command.
3. *Perform* performs a non-local service invocation (e.g. an http *get*) from the terminal. Network operators can use this to measure service performance across a statistically significant number of terminals.

While our present extension to the SyncML command set is restricted to these operations, we envisage future extensions to facilitate service management.

4.3 Guarded Commands and Output Redirection

At the finest level of granularity are policies that govern the execution of individual SyncML commands (or command blocks) within the action routine. However, we tend to not view these in the same way as the policies described in the previous section, as these are part of the functional definition of the action routine and cannot be viewed as orthogonal to the script function. Commands governed by such policies are also referred to as *guarded* commands, and the policy pertaining to the command as the *guard*. A battery-level *guard* might govern a command that fetches a number of terminal attributes. This guard would prevent the command from executing if the battery-level on the terminal is below a certain threshold. Figure 2 shows a guarded command with a battery guard**.** The standard SyncML command is prefixed by a guard element that constrains the GET to execute only when the battery level on the terminal is above the threshold. Guards may be used to protect terminal or network resources, and to coordinate with external events.

Traditionally, the results of a SyncML session are returned synchronously to the caller when the command(s) are complete. In Symple, we allow a more flexible output communication by allowing the results of partially executed scripts to be exported to URLs external to the terminal, and by allowing commands within an action routine to post their results to different URLs. *Redirect* is an output redirection primitive in Symple that delivers the output of a command to the appropriate URL Redirect will support a variety of standard protocol prefixes including http, sockets and RMI.
Allowing external access to partial script results allow the operator greater visibility into script execution and greater flexibility in controlling the script. The operator may decide to shut down an otherwise expensive and long-running script based on viewing the partial results, or he might decide to take alternative actions based on what is observed. Partial result availability is also useful in scaling the Symple paradigm to concurrent multi-terminal operations, where the partial results of a running script on one terminal may cause side effects on another.

5 Implementation and Usage Experience

MobiMan has been used in a number of operator management scenarios, of which one pertaining to wireless system performance management is described here. A wireless network operator may be interested in performance metrics such as average latency experienced by users in a particular geographic area, perhaps as a way to validate the service level agreement with an enterprise customer.

Latency can be characterized by the time taken to download a HTML or WML page from a remote server. To measure latency over multiple devices and a statistically meaningful period of time, the operator can compose guarded synclet (see Figure 3), and can push it to of the mobile device fleet.

The synclet will execute at the time specified by the date attribute and will connect to specified *URI* to download HTML page. It will repeat this operation for times as specified in *repeat* attribute. This synclet will average these latency measurements and send it to operator server. The server correlates the values obtained from number of mobile devices over period of time at different times and can collate the data to make decisions about re-provisioning the network.

```
<Synclet>
  <Guard>
     <Attribute> Signal_Strength </Attribute>
     <Condition> GREATER_THAN </Condition>
     <Threshold> 20 </Threshold>
  </Guard>
  <Schedule>
     <Date> 01028526240000 </Date>
     <Period> 1 </Period>
     <Repeat> 20 </Repeat>
     <URI> http://www.yahoo.com/index.html </URI>
     <Item>
        <Target>
           <LocURI>
              ./Sync/DM/Performance/Network/Latency
           </LocURI>
        </Target>
     </Item>
  </Schedule>
</Synclet>
```

Fig. 3. Synclet with guarded commands for latency measurement

Table 1 details our experience in running performance management Synclets over a GSM network on a Java-enabled mobile handset. The data shows a total time of about 30 seconds for Synclet loading and execution. While this number is acceptable, it could benefit from improvement. Better networks will reduce this latency, while Synclet bundling allows latency to be amortized across multiple management operations. Synclets can be encoded in Wireless Binary XML (WBXML) instead of plain XML representation that can significantly reduce size of data sent (and hence time) over wireless network. WBXML is a binary format compact representation of XML that encodes the structure and content of the document entities while removing meta-information contained in document.

The Micropod, policy engine along with SyncML engine and parser occupy 100K on MIDP KVM with peak runtime memory consumption of 230 K. Local runtime operations on the mobile device (e.g. Synclet parsing) make up about 25% of the total cycle-time, and will improve as Moore's law increases the horsepower dedicated on handsets to data services. Overall, the numbers indicate the viability of a Synclet based approach for today's handsets, and its growing value with handset and network evolution.

Table 1. Synclet Execution. The phases, and some performance measurements

	Synclet Operation flow	**Size of data sent** (bytes)	**Time taken to send data over GSM network** (seconds)

1.	**Creation of SyncML package 1** Server initiates SyncML Engine on mobile device (e.g.. by SMS push). SyncML Engine creates SyncML package with capabilities data such (e.g. Manufacturer/ Model name. and credentials.	1185	3.3
2.	**Getting Synclet from Server** Mobile device sends SyncML package DM Server. Server parses the package, checks for credentials and synclets that need to be sent to the device. Server returns new SyncML package containing synclet (HTTP reply).	1822	18.5
3.	**Invoking Synclet** SyncEngine parses/extracts Synclet transfers to Policy Engine. Policy engine verifies synclet safety and semantics, conditionally schedules Synclet. Synclet reception status sent back to server.	745	8.2
4.	**Status/Result of Synclet** Server responds to synclet receipt status message.	595	11.3
5.	**Status of Synclet** Mobile device parses SyncML message and continues to process synclet.		2
6.	**Synclet Execution** Scheduled Synclets are executed in separate thread. For long-running synclets (e.g., latency monitoring), steps 4 and 5 are repeated. Results sent back to server.		

6 Intelligent Distribution of Tasks in MobiMan

Thus far, the MobiMan architecture has focused on the design and deployment of *Synclets* as a means for intelligent *scheduling* of tasks at the targeted terminal. However, the richer predicates offered by Synclet-enhanced terminal management also enables operators to define tasks that span multiple terminals and involve complex orchestration policies for successful completion.

This functionality is supported by a *SmartCloud* extension [NA03] that integrates a tuplespace-backend (provided by the **Mojave**[3] system [VL01]) into the MobiMan server infrastructure. The SmartCloud extensions provide three core mechanisms:

[3] The Mojave System developed at Motorola Labs provides a tuple-space based agent architecture where task agents can dynamically clone/relocate to the agent container that best facilitates task completion.

1. **Intelligent TM Server Selection.** Current TM operations assume the existence of a centralized server whose identity is known to the TM client. This raises performance and reliability concerns as the server becomes both a bottleneck and a single point of failure. Instead, we envision a "multi-server" approach where multiple TM servers exist, any of which are capable of delivering the task to the device. The TM servers could be co-located as part of a "server farm" or could be in deployed as individual "kiosks" in high-traffic areas such cafeterias, banks and airports. Choice of server for task dispatch is now based on *opportunity*, i.e., the association[4] of a TM client with a particular server causes that server to register with the SmartCloud as the dispatcher for that TM client. If a client associated simultaneously with two dispatchers (cellular and short-range), the dispatcher with better QoS criteria (e.g., higher bandwidth) is chosen.
2. **Task Dispatch Priority Escalation.** Intelligent server selection raises the question of how long the SmartCloud should wait for the right "opportunity". Operators (and users) may prefer the user of the quicker, more reliable short-range network for TM operations; however, users may not always be within range of a suitably equipped server. The SmartCloud extension solves this problem by using an "escalation policy" for task dispatch. Operator submit tasks with an appended deadline for task dispatch.. The SmartCloud will hold on to tasks, waiting for the *best* dispatch opportunity; however if a specified deadline is near expiry, the SmartCloud becomes more aggressive, opting to use the *first* opportunity it sees.
3. **Automated Coordination Through Aggregation.** While the multiple-server model improves the distribution of TM tasks, it can complicate the coordination of complex tasks that span multiple terminals (e.g. an operator request for 100 terminals in the "847" code to respond with bandwidth availability information). Here, the single operator request actually translates into multiple terminal-specific tasks – each being dispatched to the target over a potentially different server. Monitoring the tasks, coordinating follow-up actions and returning a unified result to the operator can become a logistics nightmare. The use of a SmartCloud backend alleviates this problem by providing a common backend "shared memory" structure that can be used to aggregate results from various tasks and automate the firing of follow-on tasks such as request-termination, and result-display.

By integrating the SmartCloud backend into the MobiMan server infrastructure, we achieve three objectives: *flexibility, efficiency* and *ease-of-use*. Operators are now required only to design the task using Synclet semantics to define the appropriate criteria for task execution. The MobiMan system ensures that the tasks are delivered within deadline – and in the most effective manner – to the targeted terminal(s). It also *automates* the monitoring and completion of coordinated multi-terminal tasks thereby reducing the burden on the operator and minimizing opportunities for operator error.

[4] An association (client-server communication) could happen over a wide-area (e.g. cellular) or a short-range (Bluetooth, Adhoc WiFi) network.

Furthermore, because the SmartCloud extensions are server-centric, they add minimal resource or execution overhead to the resource-constrained mobile terminals.

7 Related Work

A number of lightweight programming languages [LL01] provide design examples that Symple tries to emulate. SIMSpeak [KM01] aims to provide programmability to devices whose only programmable component is an extremely limited “smart card” supporting the Javacard specification. An on-card interpreter supports a simple stack-based language that uses registers instead of variables to save space. Similar choices of language primitives are made in 3GPP's USAT specification [3GPP]. Mobile code is pushed to the device via the short-message service (SMS) and security is handled partly in the device and partly in a network gateway. MobiMan supports an SMS based script dispatch in a manner similar to SIMSpeak, but has the luxury of living in a slightly less constrained Java environment than the Javacard.

The SNMP world has at least two proposals that augment the classic client-server SNMP model with scripted agents. The SNMP mid-level manager [SMLM] proposal includes a scripting language proposal called SNMPScript that is used to specify and distribute scripted agents to managed nodes. SNMX [SNMX] is another script proposal, although an SNMX interpreter weighs in at a “chunky” 400KB. A size that would be somewhat taxing on today's J2ME powered mobile wireless devices.

Sloman [MS98] and others have made a case for policy-based systems management as a means to change management policies without changes to management agents. Some of their proposed primitives (e.g. positive obligation policies) resemble those proposed in this paper. However, Symple adopts a lightweight policy framework to cater to resource-constrained terminals, includes device states as policy predicates, is tightly integrated with SyncML, and supports multi-terminal policies via a network-resident coordination infrastructure.

Heidemann [HS00] proposes a Cron-derivative (based on Xcron [GK99], another cron derivative) for intermittently connected laptops, and articulates intermittent connectivity issues similar to those discussed by us. However, the policy language here focuses on *time*-based policies, and has no support for policies based on device state.

8 Conclusions

MobiMan aims to provide complex scheduling primitives to wireless systems management, while operating within the limitations of resource-constrained Java devices. So far our experience has been encouraging, showing that a fairly sophisticated set of capabilities can be supported on a wireless terminal. Easy authoring of coordinated operations across large sets of terminals (e.g. a cellular region) remains a challenge. This requires application infrastructure support in the network and interaction primitives to be defined for greater interplay between network controller objects and the currently executing process on a particular terminal. Emerging smart wireless terminals that support *pJava* (or J2ME/CDC) provide another systems management innovation opportunity, as they are substantially more resource-rich than the *kJava* (J2ME/CLDC) devices targeted in this paper.

References

[GK99] G. Kuenning, *A Cron Daemon for Portable Computers*, UCLA Computer Science Department Technical Report UCLA-CSD-990044, Sep 1999

[GY95] G. Goldszmidt and Y. Yemini, Distributed *Management by Delegation*, in Proc. of the 15th ICDCS Conference, IEEE Computer Society, pp 333-340, 1995

[HS00] J. Heidemann and D. Shah, *Location-Aware Scheduling with Minimal Infrastructure*, In USENIX Conference Proceedings, pp.131-138, Jun 2000

[JN01] A. Jonsson. and L. Novak, *SyncML – Getting the mobile Internet in sync*, Ericsson Review No. 3-2001, pp. 110-115

[KD99] K. Dulaney, *TCO for PDAs: Higher than Expected*, Strategic Planning, SPA-08-7900, Research Note, Gartner Group, July 1999

[KM01] R. Kehr and H. Mieves, *SIMspeak – Towards an Open and Secure Application Platform for GSM SIMs*, in Proceedings of the Intl. Conference on Smart Cards, E-smart 2001, Lecture Notes in Computer Science, pp. 135-149, Springer 2001

[LL01] MIT Lightweight Languages Workshop, 2001, http://ll1.mit.edu/

[LS01] D. Levi and J. Schoenwaelder, *Definitions of Managed Objects for the Delegation of Management Scripts*, IETF Network Working Group RFC, Aug 2001, www.ietf.org/rfc/rfc3165.txt

[MS98] M. Sloman, *Policy Based Management of Telecommunication Systems and Networks*, First UK Programmable and Telecommunications Workshop, HP Labs, 1998

[NA03] N. Narasimhan, S. Adwankar and V. Vasudevan, *SmartCloud: Automated, intelligent task distribution in MobiMan*, Internal Draft, Motorola Labs, 2003

[OS96] O. Shivers, *A universal scripting framework, or Lambda: the ultimate "little language"*, in Concurrency and Parallelism, Programming, Networking and Security, Lecture Notes in Computer Science, pp. 254-265, Springer 1996

[SD02] *SyncML Representation Protocol Device Management Usage*, version 1.1, at http://www.openmobilealliance.org/syncml February 2002

[SM02] *SyncML Data Synchronization and Device Management*, official website http://www.openmobilealliance.org/syncml

[SMLM] *SNMP Research: Mid-Level Manager (MLM)*, White Paper, SNMP Research, http://www.snmp.com/products/mlm.html

[SNMX] SNMP Frameworks, Inc., *The Simple Network Management Executive (SNMX) Scripting Language* , http://www.snmx.com

[3GPP] 3GPP Technical Specification Group Services and System Aspects: USIM/SIM Application Toolkit (USAT/SAT), Doc# 3G TS 22.038 v5.2.0 (2001-02)

[VL01] V. Vasudevan and S. Landis, *Malleable Services,* International Journal of Software Engineering and Knowledge Engineering, Vol, 11, no. 4, pp. 389-406, 2001