System management with IBM mobile systems remote
A question of power and scale

Rellermeyer, Jan S.; Osiecki, Thomas H.; Holloway, Ernest A.; Bohrer, Patrick J.; Kistler, Michael

DOI
10.1109/MDM.2012.53

Publication date
2012

Document Version
Accepted author manuscript

Published in

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
System Management with IBM Mobile Systems Remote - A Question of Power and Scale

Jan S. Rellermeyer   Thomas H. Osiecki   Ernest A. Holloway   Patrick J. Bohrer   Michael Kistler

IBM Austin Research Lab
11501 Burnet Rd.
Austin, TX 78758

Abstract—The rise of the app revolution has brought small, simple, and affordable software tools to users for mastering mundane tasks with the help of mobile devices. With the success of the app paradigm, there is an increasing demand for applying the same spirit to problems which have traditionally been the domain of enterprise-scale solutions, e.g., system management. The challenge for building such communication-intensive applications, however, is to gather the health and performance data of a large number of machines in an agile and responsive fashion while being restricted by the scarce resources and severe power constraints inherent to mobile devices.

In this paper, we present a system architecture that tackles this fundamental challenge by making data freshness an explicit concern and allowing the application to express its freshness requirements in a fine-grained way. The application runs atop a generic data cache and collection engine that fetches fresh data from the management endpoints based on these requirements. We evaluate the performance and power consumption characteristics on a broad range of Android-based mobile devices and show that this approach increases the responsiveness of the application while at the same time reducing the power consumption.

I. INTRODUCTION

Smartphones and tablet devices are increasingly becoming the universal access point to the world of connected devices. Industry projections show network-connected personal devices, such as smartphones and tablets, outnumbering traditional personal computers in the very near future [1]. An equally important factor is the new paradigm of software development for these devices. Whereas applications for traditional computers tend to be feature-rich, complex, highly configurable, universally usable, and in essence as versatile as the personal computer itself, the app ecosystem around the latest generation of mobile devices is exactly the opposite. Most apps are small, specialized, and affordable programs designed for ease of use and seamless interaction. The first generation of mobile apps went along with a democratization of software development since these apps typically targeted mundane problems that could be solved by semi-professionals. The demand, however, is increasingly going towards using mobile apps for much more complex tasks; we are at an interesting point in time where app agility meets enterprise software.

One example is the area of system management, in which IBM has been deeply involved from both a hardware and software perspective for several decades. Monitoring server machines in data centers is a task that benefits greatly from the always connected paradigm of modern mobile devices. The ability to retrieve status information at any time from any location combined with the possibility to perform the most frequent actions from the mobile device simplifies the task of system management. While there has been considerable effort to managing enterprise mobile devices from a server [2], here we consider the reverse approach of utilizing mobile devices to manage enterprise servers.

There are several challenges in developing a mobile software solution to system management. One is clearly the design of a user interface (UI) to display a large amount of data in a concise and comprehensive way given the limited screen real estate of mobile devices. Another one is the large amount of communication involved in gathering the health and performance data from management endpoints in close to real time. Because mobile devices are resource-constrained and run on battery, energy-efficiency is paramount for mobile applications. Network communication, however, is one of the dominating factors of power consumption on modern smartphones [3] so that extensive use of network resources is considered prohibitive. Therefore, the key research challenge is how to deliver fresh data at a rate so that system malfunctions can be detected in a timely manner while still being power-efficient. IBM Mobile Systems Remote (short IBMRemote) is an experimental app for smartphones and tablets developed by IBM Research for managing IBM’s server line of products, especially IBM BladeCenters. The philosophy behind it is to be the mobile system status summary, updating itself periodically, and providing detailed information at a fingertip. As such, it can be considered the antipole to the feature-rich, universal, but at the same time highly configuration-intensive enterprise solutions to system management. By separating user interface, data model, and communication mechanism, we are able to carefully balance the fidelity of the information provided and the costs attached to keeping it up to date.

The contribution of this paper is twofold: First, we present the View-Collect-Engine (VCE) pattern in, a generic architecture for gathering and monitoring data from connected devices (section III). We cache the data on the local device and use an asynchronously running data fetching engine to update the cached data periodically. The application uses a publish-subscribe interface to the cache despite the fact that the data needs to be pulled from the data sources. This
not only simplifies the design but also layers responsibilities in a way that is amenable to optimization. By choosing an optimized implementation of the design, we are able to reduce battery consumption by about 20% over the commonly used multi-threaded approach to handling communication and fetching data. Second, we introduce an explicit notion of data freshness and provide the application with a way to specify its requirements to the engine. Thereby, the engine can optimize its network communication patterns and save power within the given constraints. By carefully softening the quality of service in a way that does not adversely affect the user experience, we are able to reduce power consumption by 17%, as we show in section IV of the paper. We have applied the proposed architecture to our IBMRemote app and evaluated it on different Android devices (section V). The results show that the application is able to monitor over one hundred machines simultaneously over multiple hours of continuous runtime.

II. MANAGEMENT ENDPOINTS

Management endpoints are entities in a system management architecture that directly monitor or control one or more resources, commonly physical computer systems. A management endpoint may provide a graphical interface directly to users and/or data connections to other system management components. It is common to have systems management topologies consisting of a hierarchy of connected management endpoints as well as management presentation software. These connected components then notify each other of changes either by polling or asynchronous messaging. IBMRemote is an application that connects to a specific type of management endpoint contained in IBM’s BladeCenter®(BC) [4] systems. This app is representative of a growing number of mobile applications that are being used to manage a variety of endpoints.

Asynchronous messaging assumes a common messaging protocol as well as a stable connection between these endpoints. Polling is often employed on these endpoints to avoid dependencies on more sophisticated communications infrastructure (e.g. pub/sub mechanisms). Endpoints may allow polling through a variety of protocols (SMTP, IPMI, SSH, etc.) to support a range of clients. However, polling introduces clear inefficiencies in terms of both execution overhead and network bandwidth utilization. These communication mechanisms are especially important to applications that monitor remote state. With systems management software, there is a priority put on communicating status related to the health of a system being monitored. If a problem is found, much of the other state is retrieved merely to paint a clearer picture of the overall state of the system that is having a problem. Management endpoints offer a means to send out-of-band messages via email or SMS to notify a registered user of a problem. While these uni-directional messages can help get a user’s attention, they do not provide any assistance in resolving the problem. These alerts can also become too numerous and redundant to obscure actual problems within the volume of messages.

Mobile applications provide a unique opportunity to both alert users of problems while also providing them with an interactive interface that can be used to further ascertain or resolve the problem.

While there have been solutions proposed surrounding subscribing to a remote feed of information and publishing updates from a source to a number of remote subscribers, these techniques have not been applied to the unique challenges faced by mobile platforms in monitoring legacy management endpoints. Mobile applications typically provide fast startup time, prompt notifications, and fast shutdown times. Unlike their desktop counterparts, mobile applications experience a much shorter (though more frequent) period of use. These constraints in addition to the challenges of polling or being notified of remote state changes, provide new application challenges when running management and monitoring software on mobile devices.

III. THE VIEW-CACHE-ENGINE ARCHITECTURE

Despite being initially targeted to BladeCenter systems and the AMM proprietary protocol, IBMRemote was from the beginning planned to be extensible to other types of machines and protocols. Therefore, we are interested in not just managing BladeCenters with mobile phones but a generic re-usable solution suitable for a wide range of similar problems. In particular, it was a requirement to decouple application-level UI code as much as possible from the communication code and the specific protocol implementation so that other protocols, even those supporting different patterns of interaction, can easily be plugged into the system. The result of this effort, the VCE architecture, is based on a set of design principles that provide a sufficient separation of concern to be general and reusable, scale to allow monitoring of hundreds of management endpoints simultaneously, and fulfill the requirements of energy-efficiency. The application is composed of a set of Views (top of Figure 1) which display Data Items or values derived from the data items. A data item is an atomic unit of data that can be observed from the management endpoint. Examples of data items are the serial number of a blade server or the fan speed of the left fan of the chassis. The views do not directly interact with the management endpoints but instead express their interest in being notified about changes of data items through subscriptions registered with the data cache. Since many views share their interest in the same data items, there is a local data cache (middle of Figure 1) against which the views operate. Therefore, data items that were already retrieved for a different view do not have to be retrieved again as long as they are still fresh enough. The data cache maintains the subscriptions and all data items that currently have subscribed views are periodically updated through the Engine which initiates connections and runs the actual network protocols (bottom of Figure 1).

A. Views

A significant challenge in managing a large number of endpoints from a mobile phone is the visualization and ag-
Different views, however, typically share at least some data limiting factor in terms of battery runtime of the mobile device. Communication-intensive operation which is a significantly complex graphical representation of facts (e.g., the blades in the right screenshot in Figure 2) to an update of a textual element (e.g., the machine names and identifiers to concrete protocols and commands is entirely encapsulated in the engine since it is protocol-dependent and remains opaque to both the views and the data cache. Once a data item has been resolved by the engine, however, the data cache keeps the last version of the command that was used to fetch the data for the engine to avoid resolving the item again.

The views are entirely event-driven and only react to changes of data items on one hand and user interaction on the other. Therefore, they mimic the views in a classic Model-View-Controller [5] pattern. The result of an update to a data item depends on the application and can range from a simple update of a textual element (e.g., the machine names and types in the left screenshot in Figure 2) to an update of a complex graphical representation of facts (e.g., the blades in the right screenshot in Figure 2). Subscriptions can be added and removed at any time by views, e.g., as a result of a user requesting more or less detailed data. When the user navigates to a different view, all subscriptions by the current view are canceled. When subscribing to a data item, the view provides a callback object which is executed whenever there is either an update for the data item or a failed attempt to update the data item. In addition, the view can provide a method that is invoked when the engine begins to request an update for a data item so that the UI can visualize the fact that an update is in progress (see the spinning wheel in the left part of Figure 2).

**B. Data Cache**

Fetching fresh data from the management endpoints is a communication-intensive operation which is a significantly limiting factor in terms of battery runtime of the mobile device. Different views, however, typically share at least some data items. For instance, in IBMRemote many views implicitly depend on the topology information acquired from the management endpoints. Therefore, it makes sense especially for mobile applications to cache results and only reach out to the network if the cached information is outdated. Furthermore, caching avoids repeatedly fetching data when the user quickly looks at different view and then navigates back.

In our implementation, every data item carries a timestamp that indicates when the last update occurred. When the data cache receives a new request for a data item, the default policy is to return the value currently in the cache regardless of its timestamp, but also request fresh data if the timestamp has exceeded a certain data life time. The timestamp is visible to the application so that each view can decide to either use outdated items, perhaps visually marked as outdated, or to portray the data as unavailable until an update is received. In IBMRemote, the main use case for allowing old data is when running entirely disconnected or showing information about machines to which the connection has been lost. There is still value in showing the state of the machine until the point where the disconnection happened as long as it is apparent to the user that the data does not reflect the current state.

Besides the caching of values for data items, the data cache also acts as a cache for the engine. Resolution of resource identifiers to concrete protocols and commands is entirely encapsulated in the engine since it is protocol-dependent and remains opaque to both the views and the data cache. Once a data item has been resolved by the engine, however, the data cache keeps the last version of the command that was used to fetch the data for the engine to avoid resolving the item again.

Once the engine has acquired new data through the network, the updates are sent to the data cache which maintains a list of all subscriptions per data item. Therefore, it can publish the update to all subscribers. In some cases, an update for a single data item will yield more data than requested. For instance, some of the higher cardinality data items cannot be queried individually but the entire array is returned as a result of a request for a single item. The cache is in charge of demultiplexing such multi-value responses and sending the corresponding updates to the application.
C. Engine

The purpose of the engine is to decouple communication from the data model and encapsulate the mechanism to update the data periodically. This might involve—as in our case—active polling or can simply be the management of subscriptions or leases. The key design choice is that the engine operates entirely autonomously and asynchronously, comparable to a DMA engine in computer systems. In IBMRemote, we have implemented the engine in a non-blocking single-threaded style. The remainder of the application is passive and only reacts to user input (thereby driven by the platform UI thread) or to callbacks (hence driven by the engine thread). Data cache and engine are coupled through a request queue. The engine thread uses a selector and, based on the availability of data, executes three main phases. The first phase (destruction) checks all existing connections for protocol-level timeouts, the second phase (construction) checks the queue for new update requests, and the third phase (connection handling) handles all connections that are ready according to the selector. When all commands are processed, the thread performs a peek on the queue for any pending requests for the current machine. If a new request has arrived since the connection has been opened, it is dequeued and the commands continue to be processed over the same connection. This was implemented to support traffic shaping for power optimization. Failures in command processing, exceptions, and connection issues are signaled to the data cache and can be made available to the views.

IV. Freshness of Data

In the pure, unoptimized VCE architecture, the engine delivers new data to the cache (and in turn to the subscribed application views) by periodically polling the management endpoints. In the case of monitoring computer systems, freshness of data is particularly important. It has to be ensured that data related to important events is shown in a timely manner. This naturally requires a short polling interval, which increases the network bandwidth and battery consumption. However, not all data items shown in the app relate to critical indicators and underly the same freshness requirements. Many data items (e.g., the serial number of a machine) can be considered almost static and are amenable to much larger update intervals. Such optimizations based on application-level knowledge, if exploited at all, typically result in ad-hoc solutions. In contrast, we decided to turn data freshness requirements into a primary concern of the architecture by making them an explicit part of the subscription and thereby enabling the application to express its requirements to the cache and engine. The application can define one or more base update intervals and freshness requirements as dividers of a base update frequency. For example, if the base frequency updating every 30 seconds, it can define a HIGH requirement with divider 1 and a LOW requirement with divider 4. Then, the contract is that a data item subscribed for a HIGH update interval will receive an update at least every 30 seconds whereas a subscription with a LOW update requirement at least every two minutes.

To evaluate the benefit of relaxed data freshness requirements on power consumption, we use the main view of IBMRemote (see left side of Figure 2) configured with 10 machines and monitor battery power levels and 802.11 G WLAN radio for communication. We let the application acquire a wake lock from the platform to avoid the phone from suspending the app and we dim the screen to reduce its impact on the total power consumption. We compare the unoptimized baseline implementation consistently operating with a 30 seconds update interval with two levels of optimization. The medium optimization level reduces the update frequency of the connectivity check to 2 minutes. For the application, this means that a management endpoint that has become unreachable will not be contacted again until 2 minutes have elapsed. In addition, the update frequency for the firmware, system type, and machine label (hence three out of the four data items subscribed for in the main view) is set to the same 2 minute data freshness requirement. Only the health status, responsible for setting the small icon on the left bottom side of each machine entry in Figure 2 has the high data freshness requirement and is updated with the full 30 seconds frequency. For the high optimization level, we set the data freshness of firmware, system type, and machine label to 6 minutes—since we expect them to only change in exceptional situations—but left connectivity check (2 minutes) and health status (HIGH, 30 seconds) unchanged.

Figure 3 shows the resulting network bandwidth consumption (sent and received data) over time on the HTC Evo 3D mobile phone. The medium optimization level yields substantial savings in the volume of data transferred. Additional savings can be achieved through the higher optimization level when setting the almost static data items to a high update interval. Figure 4 presents the effect of freshness on energy consumption. In this realistic setup, giving the application the opportunity to express its data freshness requirements yields a 3% absolute or 17% relative energy saving per hour when running at the high level of optimization. Notably, these optimizations do not affect the time to detect a relevant event that might require the attention of the user since only the almost static auxiliary data is updated less frequently while the freshness criteria for the critical health data is not lowered.
V. EVALUATION OF SCALABILITY

We evaluate the scalability of the VCE architecture by measuring the performance of IBMRemote on mobile Android devices. First of all, the average update latency determines how long it takes for an update request to be sent through the network by the engine until the reply has been processed and the item in the data cache is updated accordingly. This metric is relevant for the agility of the application. Second, we measure the battery consumption per hour of continuous runtime of IBMRemote. This essentially constitutes a worst case scenario since most users use IBMRemote for shorter periods of time and let the application sleep in between so that it does not contribute to battery consumption.

In the scalability experiment, we run IBMRemote with a variable number of BladeCenter management endpoints over an extended time period and measure the average update time and change in battery level. The management endpoints used in this experiment are production systems located in Austin, TX and the two sites of IBM T.J. Watson Research Center in Hawthorne, NY and Yorktown Heights, NY. Our set of machines that we can choose from consists of 132 BladeCenters in total. However, only 38 machines give us access to the full set of information due to access control settings. Therefore, there is a discontinuity in the experiment for configurations above 38 machines in the sense the data volume exchanges is not proportionally increasing after this point. We still have to handle an according number of connections so that there is value in using the configurations beyond 38 machines for assessing scalability as long as we consider them 38+ε from a power perspective. Figure 5 shows the experimental results for the update time of IBMRemote on a Samsung Infuse 4G device with different numbers of management endpoints to monitor. As expected, the latency is volatile due to network effects but overall the application scales well for higher number of management endpoints. In this experiment, the 132 machines fully contribute to the latency since every connection needs to be handled during an update period. There is no indication that the engine created a bottleneck for configurations with a hundred machines or more.

The battery consumption for this experiment is shown in Figure 6. Notably, the difference in power consumption is small between 10 and 38 machines which indicates that our optimizations come to full effect. Again, there is no indication from the values that power consumption could render monitoring of hundreds of machines with IBMRemote prohibitive. From the absolute values, we can project a continuous runtime of about 4 hours, which is satisfactory for most purposes, especially since in practice the app is most likely used for smaller time periods. We repeated the experiment for the 10 machine and the 132 machine configuration on different mobile devices all running different versions of the Android platform. The Samsung Infuse 4G still runs Android 2.2.1, the HTC Evo 3D and the Motorola Photon run Android 2.3.4, the Google Nexus S runs 2.3.6, and the Samsung Galaxy Tablet 10.1 comes with Android 3.1. The results of the experiment are summarized in Figure 7. All devices were used with their WLAN interface except for the Samsung Infuse which we could also use with a 3G connection. The left part of the figure shows the update latencies, the right part the battery consumption. Independent of the device, the update times are consistently low (<1sec), except for the 3G connection which comes close to 1.5 seconds. As far as battery consumption is concerned, there are minor differences between the devices. On the Samsung Infuse, the Google Nexus S, and the Motorola...
update latency (ms)

<table>
<thead>
<tr>
<th>Platform</th>
<th>0</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Infuse</td>
<td>0</td>
<td>500</td>
<td>1000</td>
<td>1500</td>
<td>2000</td>
</tr>
<tr>
<td>HTC Evo 3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motorola Photon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Nexus S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy Tablet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7. Performance of IBMRemote on different Android platforms

Data freshness has been discussed in a different context by Perrig et al. [7]. Their SPINS security protocols for sensor networks use data freshness as a requirement for data integrity, thereby preventing an adversary from replaying old data.

Managing systems with mobile devices, as we envision it with IBMRemote, is a motivation that we at least partly share with other commercial products. UCSand [8] is the first Android app with a GUI for the Cisco Unified Computing Systems (USCs) monitoring and control. The app utilizes the XML-API of the UCS-Manager but, unlike IBMRemote, does not update periodically. PCMonitor [9] is a commercial product designed by MMSOFT Design Ltd. for the purpose of monitoring PCs running Windows and major Linux distributions from a mobile device. The app monitors computer resources, network performance, and allows the execution of tasks such as system shutdown and restart. However, unlike IBMRemote, which is able to act autonomously from the mobile device and directly communicates with the machines to manage, PCMonitor relies on middleware that does the actual monitoring on behalf of the mobile device.

VI. RELATED WORK

The VCE architecture is certainly not the first attempt to build data-intensive applications in an optimized manner. The Odyssey project [6] was motivated by a perceived need for adapting mobile applications to changes in the environment, e.g., the quality of the network connection. It provides a framework for cooperatively mastering this task at the operating system level while being application-aware. Hence, it follows a similar philosophy as we do by allowing the application to specify its data freshness requirements, but Odyssey is focused on ensuring data fidelity under volatile network conditions while we focus on power and performance optimizations.

VII. CONCLUSIONS

In this paper, we have described a design pattern we call VCE, where application views present data items to the user which are obtained from a cache that maintains the freshness of data items through requests to the engine that obtains the data from remote sources. We have employed this pattern in our implementation of the IBM Mobile Systems Remote system management app for both iPhone and Android devices. We describe how the VCE design pattern can be extended with explicit requirements of data freshness to enable optimization of network communication patterns to reduce power consumptions within given freshness constraints. Our experiments show how explicit requirements for data freshness can reduce power consumption by 17% and that the architecture is easily able to scale to managing more than a hundred endpoints simultaneously.

REFERENCES