Scientific Research Publishing

Journal of Software Engineering and Applications, 2017, 10, 500-528
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Towards a Technology Agnostic Approach to Developing Mobile Applications and Services

Ian Warren, Andrew Meads

Department of Computer Science, University of Auckland, Auckland, New Zealand
Email: i.warren@auckland.ac.nz, a.meads@auckland.ac.nz

How to cite this paper: Warren, I. and Meads, A. (2017) Towards a Technology Agnostic Approach to Developing Mobile Applications and Services. *Journal of Software Engineering and Applications*, **10**, 500-528.
https://doi.org/10.4236/jsea.2017.106028

Received: March 9, 2017
Accepted: June 13, 2017
Published: June 16, 2017

Abstract

Today's mobile devices and networks enable the development of novel mobile service applications. Developing such applications raises many challenges including heterogeneity in terms of mobile operating systems and APIs, service availability and scalability, and providing for the diverse communication needs of different applications. In this paper, we present an overview of the Odin middleware platform whose aim is to address these challenges. Odin utilises a surrogate-based architectural model to promote a mobile service's scalability and availability. The middleware is reconfigurable, allowing mobile applications to adapt to changing operating conditions and minimise resource consumption. It also includes an optimised communication channel that masks the complexity of interacting with mobile services over mobile telecommunications networks. Using Odin's interfaces and standards compliance, mobile applications and services with consistent communication behaviour can be easily implemented on heterogeneous platforms. Through quantitative evaluation, Odin's message-based communication primitives have been demonstrated to perform favourably with leading industry push messaging providers.

Keywords

Mobile Computing, Middleware, Mobile Applications, Push Notification

1. Introduction

Mobile device and networking hardware continue to advance at a rapid rate, providing essential infrastructure for the next new generation of mobile applications and services. However, software development kits for today's prevalent mobile platforms lack abstractions and functionality to easily and fully leverage the hardware's potential. Furthermore, mobile platforms such as iOS and Android tend to exhibit much heterogeneity, making development of application

software that has common behaviour across multiple platforms challenging.

With the increase in on-board computational resources, devices are no longer limited to playing the client role. Rather, a modern device can host a mobile service. Similarly to services on the fixed network, mobile services must publish their service endpoints and interfaces, and process requests on behalf of clients, be they mobile or otherwise.

Emerging mobile applications require rich communication abstractions. Diverse applications, including those concerned with news, sports scores, stock markets, driving [1] [2] and healthcare [3] need to disseminate updates and announcements in a timely fashion to mobile users. For such cases, push notification, which is widely embedded in major mobile operating systems, is appropriate. Push notification alone, however, is insufficient for many applications. Bi-directional asynchronous and synchronous messaging, publish/subscribe communication, streaming and peer-to-peer capabilities are also desirable. For example, in a social networking application mobile users often need to broadcast content to peers. In other cases, such as an application that monitors the well-being of operatives in dangerous environments, data should be streamed from operatives' devices to other users responsible for monitoring them.

Supporting the development of non-trivial networked mobile application software introduces several challenges:

- *Heterogeneity*. Mobile-device operating systems vary significantly in terms of the communication facilities and APIs that they offer to application developers. In addition, service interfaces provided by Cloud-based operators that enable clients to communicate with mobile applications (in particular when making push notification requests) are non-standard and proprietary.
- *Reachability*. Mobile devices are typically connected to mobile telecommunications networks (3G and 4G) whose operators impose restrictions on network use. In general, network operators do not publish the (dynamic) network addresses of devices they host, giving rise to the *addressability* problem —where there is no obvious destination for a message. *Accessibility* is a related problem whereby network operators routinely employ firewalls that filter out certain kinds of network messages (particularly those originating outside of the mobile network and which attempt to establish connection with a hosted mobile device).
- *Reliability*. Mobile applications and services differ in their needs for reliable communication. Applications like remote patient monitoring have strong requirements for data delivery, while others—for example location-updating for social networking—can tolerate some data loss.
- *Efficiency*. Given the resource-constrained nature of mobile devices, particularly relating to power supply and network bandwidth, communication facilities should aim to use only necessary resources.
- *Availability*. Unlike conventional services, mobile services are susceptible to transient availability due to their limited power supplies and intermittent network connectivity.

- *Scalability*. Mobile service applications generally need to accommodate increasing numbers of mobile users and their clients. As with conventional service-oriented applications, mobile services often need to manage bursty and unpredictable client request rates.

In this paper, we present an overview of Odin [4], a middleware platform whose aim is to address the above challenges. It simplifies development of mobile services and applications. The middleware has four novel aspects:

1) *Surrogate architectural model*. An Odin application is partitioned across surrogate and mobile-device components. The surrogate is located on the fixed (wired) Internet and mediates communication between clients and mobile devices. The surrogate can offload processing from the mobile device, promoting scalability. Similarly, in cases where the mobile device becomes unavailable, clients may still receive service from the surrogate, enhancing perceived availability.
2) (*Re*) *configurable implementation*. The middleware is based on an open component model that can be statically configured and dynamically reconfigured to meet application requirements. At deployment time, for example, the middleware can be configured with a communication layer offering particular delivery guarantees. At run-time, surrogate components can be migrated across hosts, and interconnect components (communication channels that connect surrogates to devices) can be switched depending on operating conditions.
3) *Optimised communication protocol*. A key component of the middleware is an interconnect implementation that has been developed to support mobile applications and services. The necessarily interconnect addresses the reachability problem. Based on quantitative evaluation, the interconnect has been demonstrated to efficiently transport messages when compared to the push notification solutions embedded in prominent mobile operating systems.
4) *Technology agnostic communication*. The middleware defines uniform communication interfaces that support several communication primitives. The primitives allow for applications to be developed that behave similarly, regardless of the underlying device operating system. With substitutable interconnects, applications are independent of particular networking protocols and technology. In addition, Odin surrogates expose a standards-compliant messaging interface, allowing clients to send messages in a platform neutral manner. The combination of interfaces, separable interconnects and standards compliance tackles heterogeneity.

The remainder of this paper is structured as follows. In Section 2, we elaborate on the motivation for Odin by describing three diverse mobile-service applications. In Section 3, we focus on push notification since it offers a basic messaging primitive that is ubiquitous in mobile operating systems. We continue in Section 4 by presenting Odin and outline its key functionality and capabilities. We then report on a performance evaluation in Section 5 that measures Odin's messaging performance with that of industry push services. The industry push solutions

provide a convenient means to benchmark Odin's messaging functionality, upon which higher-level primitives are implemented. In Section 6 we describe related work concerned with networked mobile applications. Finally, we conclude and outline future research directions in Section 7.

2. Motivation

Traditionally, mobile devices have played the role of service consumers, but using today's mobile hardware they are capable of hosting *mobile services*. A mobile service, like a conventional service, is discoverable and exposes a well-defined interface through which clients consume the service. Unlike services connected to the fixed Internet, mobile services allow data from in-built and external sensors, such as GPS, camera and environmental sensors, to be accessible to clients, are they mobile or otherwise. Mobile service applications thus have the potential to offer new and novel services. In this section, we introduce three diverse examples of mobile services with differing communication requirements: a media service, a remote monitoring service and a social networking service.

Figure 1 introduces the mobile media service. Over time, the service captures images and video using its device's cameras. Clients request image and video content according to specified criteria, such as time and location. In addition, clients can make requests of the mobile service to take still images and record video.

The service needs to be able to return media to clients upon request. This could be handled by either request-response messaging or publish/subscribe communication. With request-response messaging, clients would send a request message, identifying required content, which would result in a consequent response message containing the data. Alternatively, clients could subscribe to the mobile service, specifying content of interest to them, and be asynchronously notified via publish messages when the content is available.

A very different mobile service is shown in Figure 2. This service is concerned with remotely monitoring operatives who work in isolated and potentially dangerous environments, for example power company employees who maintain power lines. Such operatives regularly need to work in rural and uninhabited regions that are not supported by mobile networks. In such areas, satellite communication links must be used. Hence, operatives carry a satellite transceiver that connects to their smartphone and which provides connectivity to a central monitoring location.

With this well-being service, operatives also wear a body-area network of vital-sign sensors, which communicate with the operative's smartphone. The information acquired by the sensors is transmitted to the central location in real-time. Any anomalies in the sensor data detected by the mobile service or at the central location are processed and lead to appropriate actions, for example summoning emergency crews and directing them to the operative's location, which is made available by the service. During monitoring, central location staff might push notifications to operatives giving them advice and support. Similarly,

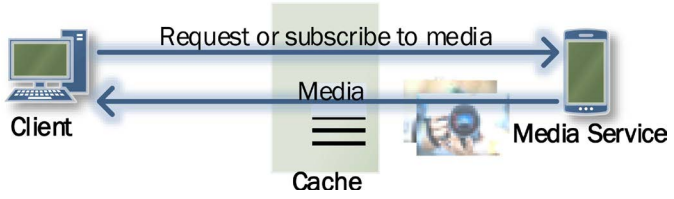

Figure 1. Mobile media service.

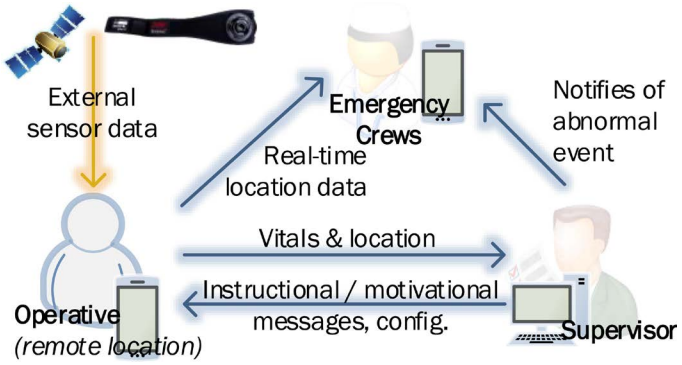

Figure 2. Operative monitoring mobile service.

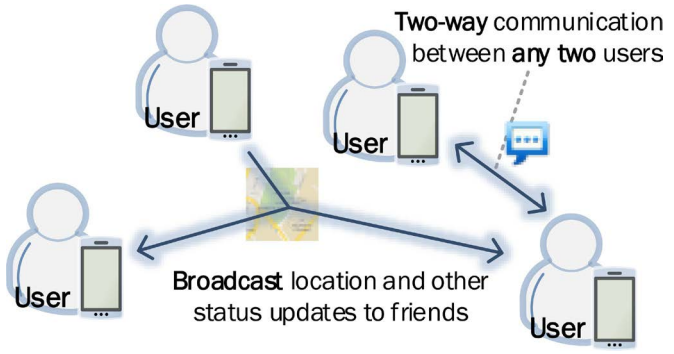

Figure 3. Social networking mobile service.

central staff might request an operative's service to take additional measurements to effectively monitor their well-being. At any time, operatives can log symptom reports, to be reviewed by central staff.

The communication primitives required for the well-being service include streaming and messaging. Streaming is required to continuously transmit vital-sign data, while asynchronous messages are necessary to exchange advisory notices and symptom reports. Synchronous messaging is required to allow the mobile service to respond with results to requests for additional health data.

The third motivating application is a social networking mobile service, shown in Figure 3. Using this service, users can broadcast their location along with other state-changes to members of the social network. The unique communication requirement in this case is for multicast or broadcast communication. With multicasting, a user's updates can be sent to selected members of the network, and, with broadcasting, all members are updated. Multicasting can be combined with publish/subscribe communication, allowing members to register for updates from particular users. Furthermore, synchronous messaging can be incor-

porated allowing users to multicast/broadcast requests that result in responses from peers.

In addition to their varying requirements concerning communication, the above services also differ with respect to some of the fundamental challenges introduced in Section 1. Of the three services, the operative monitoring service has strong *reliability* requirements. Given the nature of the service, data integrity is critical and it is imperative that there is no data loss. The mobile media service has particular *scalability* requirements since the service manages high-bandwidth data and might attract many clients. Mobile devices alone are unlikely to be sufficiently preferment to meet the needs of concurrent clients. Table 1 summarizes the key requirements of the services.

Other challenges tend to cross-cut all three services. It is desirable that they each accommodate *heterogeneous* mobile platforms, allowing the services to be developed and deployed on different mobile devices. Being mobile services, they each need to be *reachable* when connected to mobile networks. Similarly, their limited computational resources must be conserved, requiring *efficient* communication solutions. Finally, since mobile devices are susceptible to *availability* issues, additional infrastructure is required to mitigate device unavailability.

Meeting these challenges suggests the development and adoption of suitable middleware. Prior to describing our Odin middleware (in Section 4), we focus on push notification, which has been universally adopted by mobile platform vendors as an asynchronous publish/subscribe messaging primitive.

3. Push Notification

The push notification paradigm essentially involves mobile devices (smartphones) subscribing to a, typically Cloud-based, notification service. The service is responsible for pushing content to subscribers when appropriate. In support of push notification, the OMA (Open Mobile Alliance)—whose goal is to promote interoperability through open specifications—has developed a set of specifications that collectively constitute a framework that is represented abstractly in

Table 1. Mobile service applications' requirements.

Application	Requirements
Media service	Request-response messaging
	Publish-subscribe communication
	Caching
Operative monitoring service	Asynchronous messaging
	Request-response messaging
	Streaming
	Data integrity
	Specialist communication links
Social network service	Broadcast messaging
	Multicast messaging

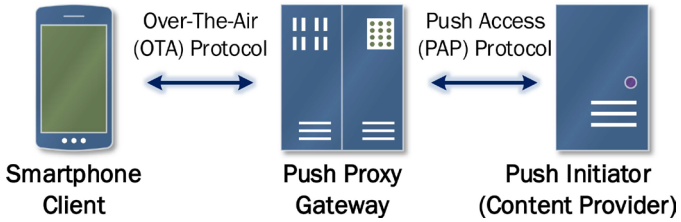

Figure 4. OMA reference model for Push Notification.

Figure 4. Being a framework, many implementations are possible that differ in terms of functionality and technology choices. The key entities in the framework interact using two families of protocols, OTA (Over The Air) and PAP (Push Access Protocol). OTA protocols ship notification messages between a PPG (Push Proxy Gateway) and a mobile device, while the PAP enables PIs (Push Initiators) to make push requests of the PPG.

At the heart of the framework lies the PPG, this plays the role of the Push service. The PPG is a proxy in that it exposes an interface to PIs and masks the complexity associated with delivering notifications to devices. To send a message to a device, the PI simply makes a call to the PPG, which is responsible for routing and delivering the message to the device. The PPG is also a gateway since it converts incoming requests into a form that can be delivered using a particular OTA protocol. In support of these roles, the PPG handles client registration, device address translation, push content transformation and transactional support for storing and forwarding push messages.

Where devices are connected to PPGs over mobile networks, the reachability problem [5] must be addressed by the OTA protocol. Reachability poses two sub-problems: *addressability* and *accessibility*, introduced briefly in Section 1. Addressability concerns the communication endpoint (destination) for a push notification; mobile network operators typically hide the addresses they assign to devices hosted by their networks—hence there is no clear destination address with which to target a push notification. Accessibility deals with how to route a push notification to a device with a dynamic point of network attachment; in general, mobile network operators use firewalls to deny connection requests originating from outside of the network through to hosted devices.

Appendices 1.1 and 1.2 further detail the OTA and PAP protocols respectively. The OMA's specifications for OTA protocols describe how existing protocols can be used to push content from the PPG to mobile devices. The PAP protocol partially addresses heterogeneity by standardizing the PAP interface, thereby supporting interoperability between heterogeneous PIs and PPGs.

Support for push notification is currently embedded in smartphone operating systems, for example Android, Blackberry, iOS and Windows Phone. Each offers a proprietary solution and an associated programming model that governs permitted content types and size of push data, client capabilities for handling incoming notifications, and QoS behavior.

To illustrate some of the differences in vendors' offerings, iOS imposes a maximum payload size of 256 bytes for a push message. Since, in general, this is

insufficient to carry application data; an iOS push notification serves as a *poke* message to a mobile device. The message carries enough data to identify a content change within the PI. In response to a poke notification, an iOS mobile application is expected to make a *pull* request of the PI to retrieve updated data. The motivation for this behaviour is that the PPG requires less storage given that messages are limited in size and that newer messages overwrite any stored messages for a given recipient. Other providers are more generous in their allowances for payload data, and consequently offer more freedom to application developers as to the content of push messages. Where push messages store application data, the PPG can directly push updated content to devices, without them needing to make an additional call to their PI. However, having PPGs store application data places greater demands on PPGs for storage. Moreover, PPGs need to be trusted if they are storing and forwarding sensitive data.

Unlike iOS and Windows Phone, Android imposes few restrictions on the type of data that can be transported in push messages. With iOS and Windows Phone, push content is associated with a semantic type that governs how the content is processed when it reaches a device. *Raw*-typed content in Windows Phone, for example, is only processed by the device if the corresponding device application is running at the time the notification is received. With Android, all incoming notifications are processed by the operating system—launching, if necessary, an application component to perform the processing.

The leading mobile operating systems and their associated PPG infrastructure also differ with respect to delivery guarantees offered, OTA protocol design that affects efficiency, and security provisioning. We direct the reader to [6] for a more thorough description of the key industry mobile platforms that offer push notification functionality.

Suffice for this paper; the heterogeneity in each of the vendor's programming models places the burden of developing a push-enabled application for multiple smartphone platforms with the application developer. Furthermore, developing an application with common push-processing behavior across multiple platforms is difficult. Moreover, since each vendor does not conform to the OMA's PAP standard, offering instead a proprietary PAP interface to their PPG, PI applications have to be implemented to use vendor-specific interfaces.

4. The Odin Middleware

An Odin application is based on the *Surrogate* model illustrated in **Figure 5**, where an application's software components are partitioned across a surrogate host, typically an Internet host on a fixed network, and a mobile device.

The surrogate host runs a *surrogate component* that acts on behalf of the *device component*, which is the part of the mobile application/service hosted by the device. The application's surrogate typically performs computationally expensive processing, caching and provides access to resources that are otherwise unavailable to the mobile device. Device application software is responsible for reading and controlling device sensors, performing local processing and updat-

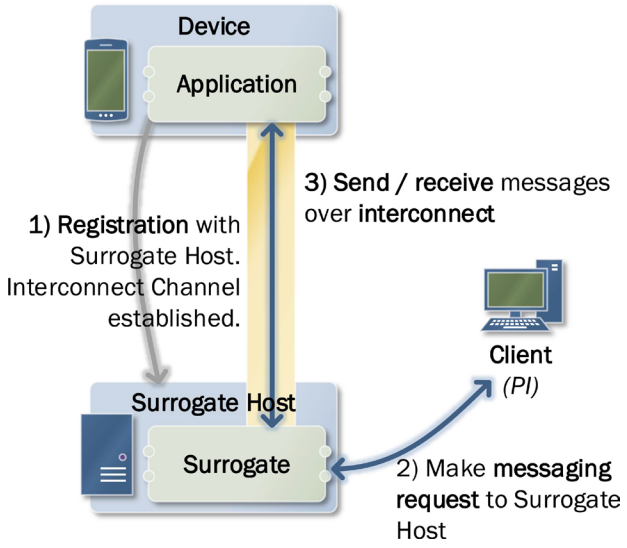

Figure 5. Odin application structure.

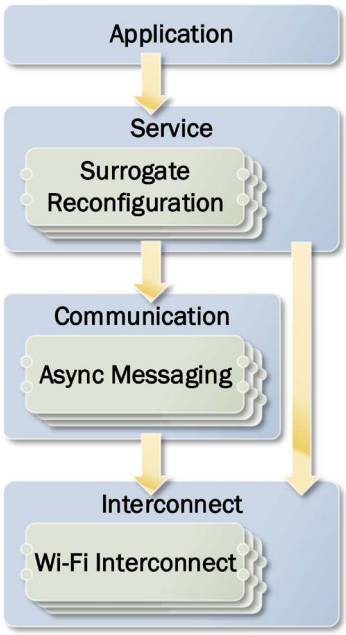

Figure 6. Odin architecture.

ing the user interface.

The middleware itself is based on a layered component model, as shown in Figure 6. The interconnect layer is responsible for providing a basic messaging service between mobile devices and surrogate hosts. Different interconnect implementations can be employed, depending on the types of network to be used and application requirements. For example, a Bluetooth interconnect provides a low-power communication channel for use when a device is in close physical proximity to a surrogate host. For mobile services connected using mobile networks, an interconnect that solves the reachability problem is required; when connected using a WiFi link, a simpler interconnect is sufficient since it needs not be concerned with barriers associated with mobile network operators. With

the remote operative monitoring service, an interconnect that uses a satellite link facilitates remote monitoring in areas not covered by mobile networks.

The communication layer is concerned with QoS and communication primitives. Similarly to the interconnect layer, this layer is configurable using different component implementations. The service layer provides functionality that is useful to mobile service applications, including caching and reconfiguration. Different caching policies are implemented by cache components. Surrogate components can be associated with caching metadata to identify methods that return cacheable content along with expiry times.

The middleware is reconfigurable with respect to the type of interconnect being used and the location of a surrogate component. Depending on the operating environment of a mobile device, its interconnect can be substituted at run-time for another. For example, a device might initially use an interconnect for connectivity to a 3G-based network. When the device enters an environment in which a surrogate host is nearby, the device's surrogate component can be migrated to the nearby host and the interconnect substituted for a low-power Bluetooth interconnect. Similarly, for the remote operative service, the satellite-based interconnect can be swapped for a cheaper mobile network interconnect depending on cell coverage. Surrogate migration is managed by the middleware to preserve surrogate state and to leave a forwarding address to clients. For interconnect replacement, the communication layer ensures that there is no loss of data when switching the interconnect. Odin thus allows for switching between heterogeneous underlying networks without compromising communication reliability.

Odin addresses the challenges identified in Section 1, as summarized in Table 2. The Surrogate model contributes to scalability, availability and reachability. For scalability, the surrogate offers a way of off-loading processing from resource-constrained mobile devices. All client requests pass through an application's surrogate component—and in many cases the surrogate can fully process the request without needing to relay it to the device component. With the mo-

Table 2. Summary of how Odin addresses identified challenges.

Challenge	Solution
Heterogeneity	Cross-platform API Adherence to OMA's PAP standard Substitutable interconnect
Reachability	Interconnect protocol that masks mobile network idiosyncrasies and device mobility
Reliability	Middleware protocol that is tolerant of network disconnection and which ensures eventual delivery of messages
Efficiency	Interconnect protocol that has a low bandwidth overhead
Availability	Surrogate model that maximises *service* availability despite *device* unavailability
Scalability	Surrogate architecture, allowing devices to offload computationally intensive tasks and surrogate components to cache data

bile media service for example, the surrogate component is implemented to use caching to serve as many requests for content as possible without involving the mobile device. Expensive image processing tasks are also performed by the surrogate component. The effect of the surrogate component being responsible for these tasks means that the mobile device is more able to process other types of requests that necessitate action by the device, such as taking images and footage.

The Surrogate architecture promotes availability by being able to mask periods when the mobile device itself is unavailable to provide client processing. The transient network connectivity of mobile devices and their limited power supplies is mitigated by employing a surrogate component. As discussed above, using the surrogate, clients can often receive processing from a mobile service without involvement from the mobile device. The Surrogate architecture can thus provide the illusion of device availability for many types of requests, when in reality it is the surrogate component of the service that is actually providing service.

With Odin's Surrogate model, reachability is the concern of the middleware implementation rather than the developer's application. Essentially, the device-side middleware is responsible for establishing a network connection with the surrogate. Since mobile network operators allow connection-requests originating within the mobile network to hosts on the fixed Internet, the requests are permitted to be made by mobile devices. The interconnect maintains the connection, restoring it as and when necessary, such as when the network address of the device changes due to its mobility. In cases where reliability is required, the communication layer operates a middleware-level protocol that tracks transmitted data and, in the event of a broken connection, ensures that data is resent over a restored link. Similarly, for the efficiency challenge, the middleware protocols have been designed to minimise overhead while providing QoS guarantees.

For the remaining challenge, heterogeneity, the middleware is ported to different mobile device types. On each mobile platform, the middleware provides a uniform interface and set of APIs that allow applications to be developed relatively quickly. Through offering the same capabilities, mobile applications and services can be developed with consistent behaviour across different devices. For push notification, mobile applications can be developed to process incoming notifications as appropriate, without being constrained by the underlying operating system support for push notification. In addition, and as discussed earlier, surrogates expose the standard PAP interface allowing PIs to make push requests without knowledge of the types of devices used to run the mobile application software.

4.1. An Interconnec for Mobile Networks

Three designs for an interconnection that is suitable for mobile services and push notification over mobile networks were considered. Messages sent from the device to the surrogate are easy to manage—since mobile networks allow devices to initiate communication with hosts located outside of the mobile networks.

Sending messages from surrogates to devices is more difficult, because of the reachability problem. We focus on the latter in describing the interconnect designs. Beyond the reliability concerns, the interconnect should use no more processing and bandwidth than necessary to conserve device resources.

1) *OMA's client-initiated scheme*. Use of SMS is to prompt the device-side middleware to initiate a HTTP connection with the surrogate. Once established, the device makes a HTTP request to which the surrogate responds by including the application message, destined for the device, in the corresponding HTTP response.

2) *Polling*. Periodically have the device-side middleware open a HTTP connection to its surrogate, requesting any pending messages. Any messages queued by the surrogate can then be transported over the connection in the HTTP response.

3) *Persistent connection*. Have the device-side middleware initiate and maintain a long-lived TCP connection with the surrogate, over which the surrogate can relay messages as and when they need to be sent.

All three designs solve the reachability problem as they involve the mobile device establishing a connection to the surrogate on the fixed network. Options 2 and 3 do not rely on additional infrastructure, such as Option 1's use of SMS. Options 1 and 2 have the advantage over design 3 that the connection is set up and quickly torn down. Obviating the need to maintain a persistent connection would lead to a slight reduction in power consumption. Option 2, however, is associated with fundamental problems since polling involves a trade-off between data freshness and efficiency. With a polling rate that is too frequent, power and bandwidth is used unnecessarily because there are no messages to transmit since the last poll. Conversely, messages may be stale when delivered using a long polling interval, resulting in an unresponsive messaging mechanism. Furthermore, excessive polling leads to scalability issues for the surrogate as many devices can be simultaneously over polling.

Our interconnect takes the form of a persistent connection. Once established, the surrogate can relay messages on demand and instantly, with no additional connection establishment overhead. In the absence of messages, the connection would be idle, and vulnerable to closure by the networking infrastructure of mobile network operators. To address this potential threat to operation, the interconnect regularly transmits a *heartbeat*, a small piece of data, which prevents the connection from being detected as quiescent. To minimize bandwidth consumption resulting from heartbeat transmission, the interconnect finds the minimum frequency required to maintain the connection and is further optimized to send heartbeats only when the connection is not being used to transmit message (as the channel is deemed active at such times).

The choice of TCP as the protocol to underly the interconnect offers more flexibility than HTTP. Once established by the device, the TCP connection supports full duplex capabilities. This is useful for devices to not only receive messages, but to send heartbeat messages and application messages, and to acknowledge message delivery. The interconnect protocol transmits heartbeats

using request reply semantics—allowing the heartbeat to double up as a mechanism to indicate a failure connection. Where the device-side middleware fails to receive a response to its heartbeat message, it can attempt to re-establish a functioning connection. In all cases of reconnection, whether due to a device's IP address being reassigned by the mobile network, the original connection being forcibly terminated, or because of arbitrary failure, the interconnect is responsible for recovery.

4.2. Communication Layer

Communication is subject to a number of QoS requirements. Messages generally need to be delivered with low latency, and eventual delivery should be guaranteed despite transient device connectivity. Furthermore, for many applications, message ordering is important—with messages being delivered in the order in which they were sent.

Figure 7 shows the states of a message, beginning with the surrogate's request to send the message. The surrogate invokes the sendAsyncMessage method, sending a JSON message consisting of the ID of the device along with any number of key-value pairs comprising the message body. sendAsyncMessage returns immediately with a MessageHandle object that may be used to query the status of the message, wait for a certain lifecycle state, or wait for a response if that message was a request message in a request/response exchange.

In the PendingSend state, the message is stored by the surrogate and scheduled for delivery. Once scheduled, the message will be sent if the interconnect provides a functioning connection to the associated device. Given a connection, the message's state is changed to Sending and an attempt is made to transmit the

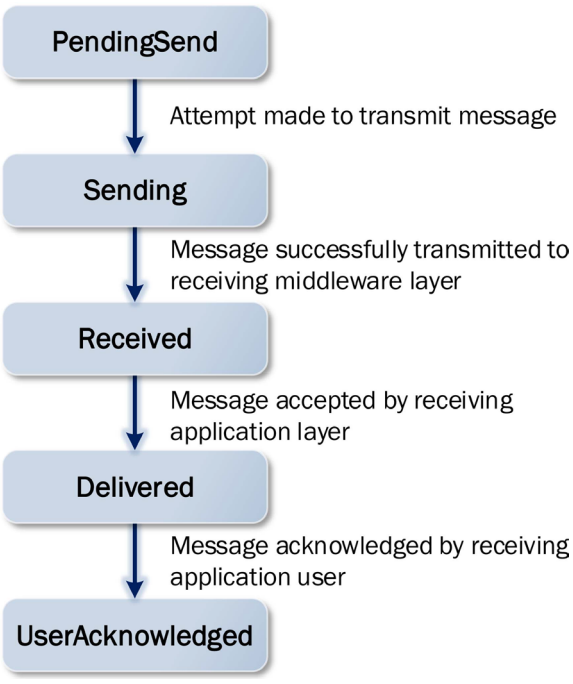

Figure 7. Message lifecycle states.

message. If the request times out or another communication error occurs, the connection is marked as broken. The message remains in the surrogate's storage; if the message was sent successfully it is stored until acknowledged, and in other cases it is stored while waiting for a connection to be resumed. Unacknowledged messages are eligible for retransmission after a specified expiry time.

Assuming the message is received by the device, its state is changed to Received, with this being communicated back to the surrogate. The device-side middleware discards any duplicate messages it receives—following retransmission by the surrogate prior to the acknowledgement being received by the surrogate. In cases where the device-side middleware cannot transmit the Received state of the message, the device will attempt to re-establish connectivity. When a surrogate receives a state update for a message of Received status or greater, it removes the message from its store.

Following receipt of a message at the device, if its sequence number is correct, it is immediately delivered to the application and its Delivered status is transmitted back to the surrogate. Out of order messages are buffered until all messages with prior sequence numbers have been received. Similarly, for multicast messages, they are buffered until all devices have received the message.

In addition to point-to-point communication, Odin supports *multicast* messaging. Surrogates can send messages to multiple specified devices. The messages are sent over a dedicated interconnect, one for each device. Like the unicast send operation, the multicast send is reliable. The middleware protocol ensures that multicast messages have been received by all devices prior to the devices releasing the messages to the application layer. In support of the OMA PAP standard, multicast messages can be replaced and cancelled atomically (*i.e.* messages are replaced/cancelled for all destined devices meaning that all of the devices process the same messages).

4.3. Multi-Platform API Support

To facilitate the development of cross-platform mobile applications, we have implemented a common API for both the Android and Windows Phone platforms. The API implementations differ based on the characteristics of the different device operating systems, but provide similar capabilities. A high-level view of part of the API, which focusing on basic messaging, for Android is shown in **Figure 8**.

When writing Odin applications for Android, developers access Odin's functionality using the OdinService class, which is implemented as an Android *service* [7]. This class exposes the interface necessary to send messages to devices and the surrogate. Hence, the surrogate- and device-side components use OdinService to initiate communication. ServiceComponent also accepts an OdinServiceCallback object, which is used to deliver messages once they have been received.

In Windows Phone, there is no equivalent to an Android service. As such, the OdinService class is implemented to use native Windows Phone threads in this

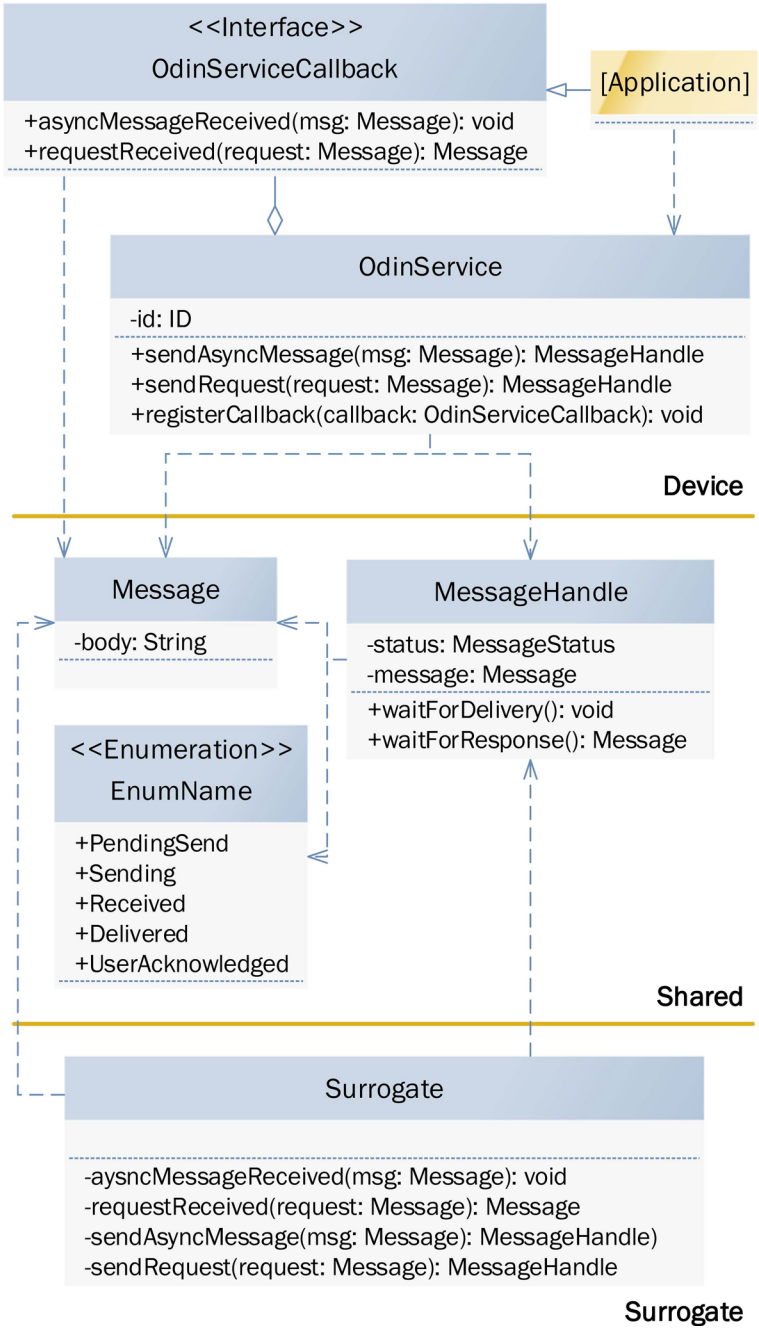

Figure 8. Overview of part of the Odin API for Android.

case. Its interface to developers is similar to Android, except that the callback for message delivery is implemented using a delegates as opposed to an interface implementation.

Lacking Android's service component, Windows Phone's support for background processing is limited. When deployed on any Windows Phone device, a background task may be permitted to run periodically, but it cannot perform resource-intensive processing since this could detract from the user's experience with the current foreground application. The background task can process incoming messages, but given limitations of the frequency at which the task can

run it cannot guarantee their timely processing. Nevertheless, the Windows Phone middleware port addresses the other QoS issues in providing for remote communication service.

In addition to message handling, Odin's surrogate-side API provides another extensibility point. By default, surrogates simply relay client requests through to the intended mobile device. However, as discussed earlier, the surrogate can add value by processing requests, where possible, without involving the mobile device, perhaps using Odin's caching APIs. In other cases, the surrogate can perform either pre-processing or post-processing logic around requests that are partially executed by the mobile device. Since surrogate components are implemented using the middleware's APIs, they can be used with any devices supported by Odin, regardless of their particular mobile device platform.

4.4. Push Notification Using Odin

With the messaging interconnect described in Section 4.1 and the publish/ subscribe primitives of the communication layer, Odin offers the means for developers to build mobile applications, regardless of their underlying mobile platform, to consume push notifications in a consistent way. Coupled with PAP-compliant surrogates, Odin offers a technology-agnostic alternative to proprietary push service providers.

When building Odin applications that use push notification, application requirements can be used to drive the behaviour of the push notification facility. This is in contrast to the industry solutions that tend to implement fixed and rigid behaviour, concerning, for example, message size constraints, limited semantic types that determine how notifications are processed, delivery guarantees and architectural model constraints—such as the poke-and-pull model described in Section 3. Rather than impose particular push notification behaviour on the application, Odin allows the behaviour to be configured to suit application needs. Table 3 identifies key attributes that developers have control over when using Odin for push notification.

As mentioned in Section 3, notification handling in Windows Phone (and iOS) is based on a type that is associated with the push message. The *raw* notifications for Windows Phone are processed by the device only if the user is engaged with the corresponding application at the time that the incoming notification arrives. Windows Phone also supports *toast* notifications, which always involve prompting the user but are processed only if the user chooses to do so. The

Table 3. Configurability for application-driven development.

Attribute	Description
Handling	How an incoming notification is processed
Type	Permitted data types for push payload
Size	Maximum size of push payload
Reliability	Reliability of device-to-PPG communication

implicit assumption with Windows Phone is that raw notifications are never of interest to applications when they are inactive, and that toast notifications are not to be processed automatically. This leaves a void in that a Windows Phone application is not guaranteed to process push notifications. With Odin, developers have the freedom to specify the way in which particular notifications are to be processed.

Whereas APNS and IBM's Message Queue Telemetry Transport (MQTT) service mandate small push payloads (256 bytes), and so force applications to conform to the *poke-and-pull* model, using Odin developers can work with messages of arbitrary size. This allows applications to include content in push messages, if appropriate, and to have freedom of choice over architectural models.

In addition, the extent of reliability guarantees can also be varied with Odin. For many applications, the *fire-and-forget* (best effort) semantics provided by APNS, MPNS and GCM is sufficient, but for others *exactly once* semantics is demanded. Configurability allows developers to make tradeoffs between reliability and cost. As higher levels of reliability are sought, protocol overhead tends to increase [6].

Through the PAP interface exposed by the surrogate (PPG), clients (PIs) are provided with a comprehensive set of operations for submitting, manipulating and tracking push requests. Furthermore, the PPG functionality is exposed in a manner that is independent of any service implementation, including heterogeneous mobile devices and communication networks.

5. Quantitative Evaluation

We conducted an evaluation to measure the performance of Odin's messaging capability over mobile networks, focusing on reliability and efficiency. For reliability, we measured the following under both normal and stressed operating conditions:

- *Message loss rate*. The proportion of messages that are not delivered in response to send requests.
- *Message ordering*. The order in which messages are delivered following send requests.
- *Duplicate messages*. Messages that are delivered more than once.

For efficiency, we measured:

- *Responsiveness*. The latency between a PI making a send request and the corresponding message being delivered to the client.
- *Bandwidth use*. The amount of data transmitted for each successfully delivered message.
- *Energy consumption*. The power required by the client device to operate the middleware.

To provide a meaningful context, we benchmarked the results against 4 push services that are available for prevalent smartphone platforms: APNS (Apple Push Notification Service), BBPS (Blackberry Push Service), GCM (Google Cloud Messaging), and MPNS (Microsoft Push Notification Service). An over-

view of these of services is presented in [6].

5.1. Experiment Setup

To gather the measurements, three types of experiments were conducted. First, a series of 2 hour experiments were conducted for each service, involving a PI making a push submission with a 256 byte payload at a rate of 3 per minute. Variables comprised network type and connectivity for the OTA link. For each service, separate experiments were run over WiFi and 3G networks. In addition, a subset of the experiments assumed a fully connected OTA link for the duration, while others involved deliberate periods of disconnection up to 30 minutes. To load test the services, a different experiment was conducted to determine the rate at which the solutions could sustain push submissions with consequent notifications being delivered to the client. Finally, 24-hour experiments were run for each service, assuming the same payload and submission frequency of the 2-hour tests, to determine energy consumption. Similarly to the 2-hour experiments, variables included the use of WiFi and 3G for the OTA connection. In cases where the client device operated for the full 24 hours the remaining battery charge was recorded; in other cases the services' operational time was logged.

Equipment used to run the experiments included four contemporary smartphones. For Odin, a Samsung Galaxy II running Android 4.0.3 and a Nokia Lumia device with Windows Phone 7.1 was used. The Samsung device was also used for GCM, and the Nokia for the native MPNS service. For APNS and BBPS, a Blackberry Torch 9860 smartphone running Blackberry 7.1 and an Apple iPhone 4S hosting iOS 6 were used respectively. Bandwidth over WiFi connections was measured using Riverbed's AirPcap wireless packet capture device in conjunction with the Wireshark protocol analyzer. For hosting the Odin surrogate (PPG), Microsoft's Cloud-based Azure service was used. The Odin middleware was configured with the interconnect described in Section 4.1 and a communication layer guaranteeing exactly-once delivery semantics.

5.2. Results

In the remainder of this section, we report on the results from running the experiments. We begin with the reliability measures and then discuss the findings for the efficiency aspects.

5.2.1. Reliability

As expected, Odin provided reliable communication. Across all reliability experiments, the message loss rate was zero; notifications were delivered in the order in which corresponding submissions were sent and without duplication. In the absence of forceful OTA disconnections, the push services also exhibited reliable communication. Of these, the BBPS and MQTT variants used in the study provide delivery guarantees while APNS, GCM and MPNS are best-effort services. During the experiments that involved deliberate OTA disconnection, all services with the exception of APNS and MPNS eventually delivered queued notifica-

tions. To reduce storage space requirements, APNS' PPG implements an aggressive overwriting policy; where a push submission is made targeting a particular client, any queued notification for the client is overwritten with that of the new push request. MPNS buffers only a limited number (32 in the study) of notifications for any given client device.

Figure 9 shows the results of service load testing. The throughput results for Odin (64 req./sec. for Android and 20 req./sec. for Windows) represent the limit at which incoming requests can be processed. Beyond this, Odin buffers notifications ensuring no loss. Odin is able to maintain relatively high throughput because, unlike the commercial service providers, its PPG does not throttle request processing.

The Android-based service was able to sustain a higher throughput because the Samsung device has a dual-core processor whereas the Nokia phone has only a single-core. The industry services are Cloud-based with provisioning responsibilities lying with the service vendors; their behavior under load differs. For APNS, making push submissions at a rate greater than the consequent notification can be dispatched results in notification loss because of APNS' overwriting policy. With throughput over 4 req./sec. BBPS experienced notification loss. GCM attempts to manage notification loss by a combination of buffering and throttling the rate at which notifications are dispatched to the target device, maintaining the rate under 10 req./sec. Buffer space became exhausted when push submission were made at a rate of 35 req./sec. Finally, MPNS employs a different tactic by blocking the PI when it sends push submission requests at a rate greater than 4 - 5 req./sec., reducing the need for large buffer space.

5.2.2. Responsiveness

Figure 10 shows the median latencies for push notification delivery. For each service, the top of the vertical line represents the 3G latency and the bottom that for WiFi. The Odin push interconnect shares the dynamic heartbeat mechanism with GCM, providing comparable responsiveness. Being proprietary, the implementation details of APNS, BBPS and MPNS are unknown, but, again, latency across all surveyed services is broadly comparable.

5.2.3. Bandwidth Use

Bandwidth used per notification received is shown in Figure 11, with data in excess of 0.25 KB being push-protocol overhead. The Odin mobile-network interconnect performs relatively well since its protocol feature-set is minimal yet sufficient to meets its quality of service guarantees. Notification channels for other services are typically used to also ship other operating system management data. With the exception of BBPS, bandwidth consumption over 3G is greater than that for WiFi. This is because over the 3G networks, connections are frequently broken and require re-establishing, incurring additional overhead. GCM operates notably less efficiently in combating the fragility of 3G connections. Unlike the other services' PPGs, the PPG for BBPS compresses push content when communicating with devices over a 3G bearer and yields a 23% reduction.

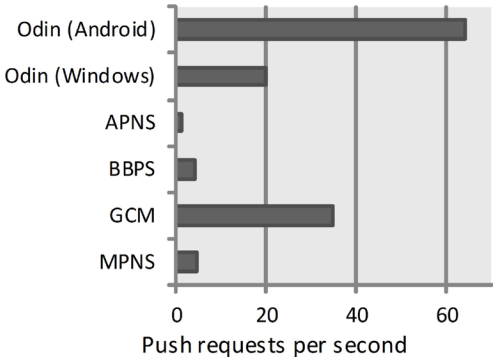

Figure 9. Load testing.

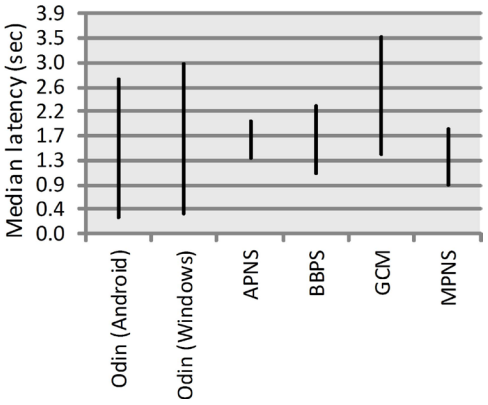

Figure 10. Median latencies.

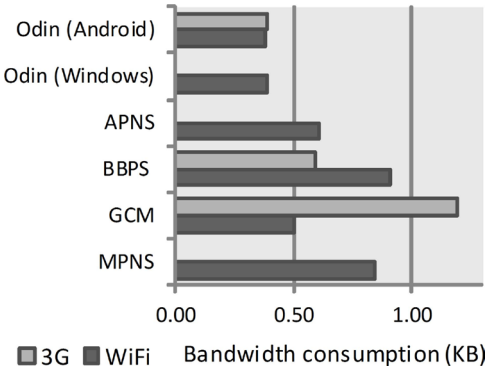

Figure 11. Bandwidth consumption.

Figure 11 omits bandwidth consumption values over 3G for APNS and MPNS because the iOS and Windows Phone operating systems lack API support for measuring data use over 3G.

5.2.4. Energy Consumption

Concerning power consumption, Figure 12 shows how much battery charge remained after 24 hours when the devices are idle. The figure also shows the remaining charge after 24 hours' operation over WiFi, processing incoming notifications with a payload of 256 bytes at a rate of 3/min. The devices used for APNS and MPNS exhausted their power supplies prior to the 24-hour period expiring—hence the zero values for Odin (Windows), APNS and MPNS. None

of the devices were able to provide 24 hours' service over 3G bearers. Figure 13 shows the operational times for all services over 3G, and, additionally, the operational times for the services that failed to operate for 24 hours over WiFi (*i.e.* the services with zero battery levels in Figure 12).

The programming models for iOS and Windows Phone are responsible for the relatively high energy consumption when using APNS and MPNS. With iOS, each incoming notification involves a brief activation of the device's screen to alert the user. Windows Phone requires that for a notification to be processed, the associated application must be running and that the user must be interacting with it—necessitating an activated screen. Activating a smartphone's screen draws significant power, hence the relatively rapid depletion of power.

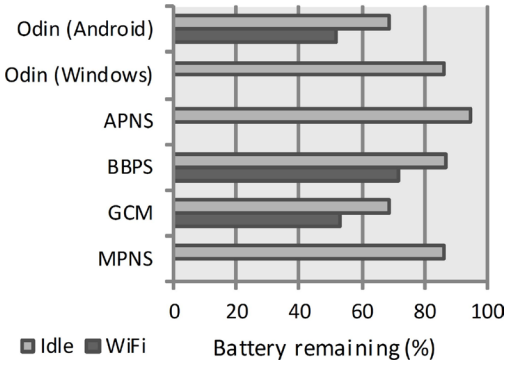

Figure 12. Percentage of battery remaining after 24 hours operation.

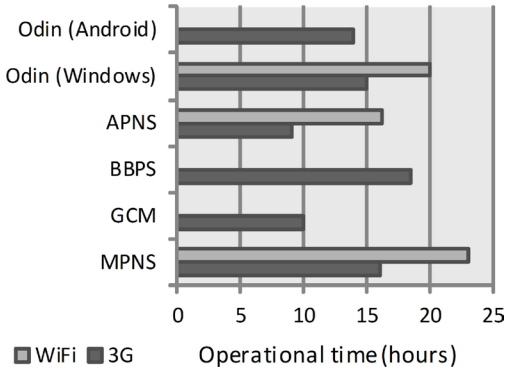

Figure 13. Operational time.

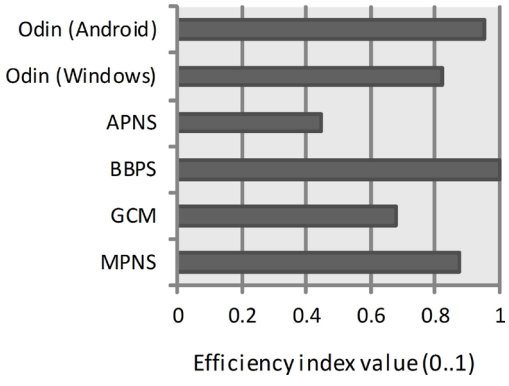

Figure 14. Energy efficiency index.

The Odin implementations for Android and Windows Phone manage power similarly to the other services that run on the Android and Windows Phone devices. Over 3G, Odin for Android performs better than Android's native GCM since the latter generates extra traffic when maintaining connectivity over 3G. Odin for Windows Phone performs comparably to MPNS. In drawing comparisons relating to the energy efficiency of the surveyed services more generally, bearing in mind device heterogeneity, we have devised an *energy efficiency index*, as shown in 3, which measures the relative efficiency of each service's messaging protocol. Intuitively, smaller differences between a solution's operational time and the hosting device's operational time at idle indicate greater power efficiency of the protocol. The efficiency index quantifies this, as a value between 0 and 1 for each solution, and assuming 3G operation, as follows:

$$T_{operational} / T_{maxOperational} \quad (1)$$

$$T_{deviceIdle} / T_{maxDeviceIdle} \quad (2)$$

$$(1)/(2)/T_{maxOperational} \quad (3)$$

6. Related Work

Approaches to supporting remote communication involving mobile devices can be classified as follows:

- Embedded middleware.
- Intermediary-based middleware
- Network-based solutions.

Table 4 summarizes the allocation of notable work by others to the three classes, and highlights which challenges from Section 1 they address.

Embedded middleware is deployed directly on host devices. Generally, embedded middleware is a lightweight version of regular middleware that is used by machines on the fixed network. Intermediary-based middleware consists of some components deployed on mobile devices, with others (the intermediaries) being deployed on machines within the fixed network. Odin, with its Surrogate-based model, is an instance of intermediary-based middleware. Network-based solutions work at a lower level, taking the form of network-layer protocols.

Figure 15 illustrates the general form of embedded middleware. Rover [8], ICE-E [9] [10], and SoapME [11] have been developed with the intention of allowing mobile devices to host mobile services. Host addresses are expected to be published via some sort of service registry, and once obtained clients can make requests of the mobile services. The communication protocols of the lightweight

Table 4. Related work by category and challenges addressed.

Class	Instance	Challenges addressed
Embedded	Rover, SoapME	Reachability, efficiency, reliability
Intermediary	MobileHost MSP CloneCloud	Reachability, efficiency, reliability, scalability
Network-based	Mobile IP, GTP	Reachability, efficiency

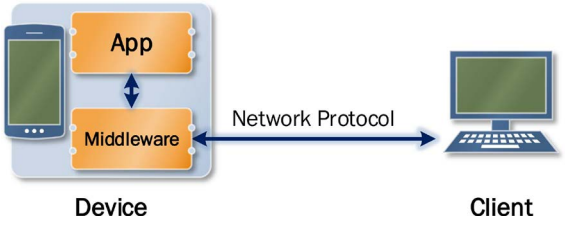

Figure 15. Embedded middleware.

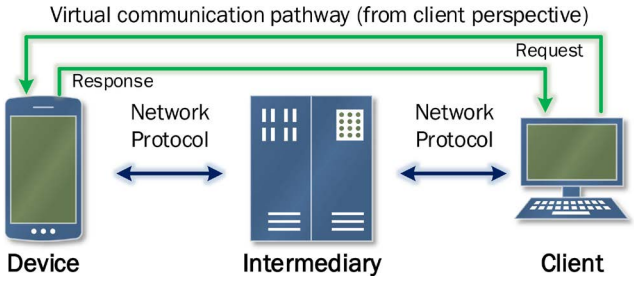

Figure 16. Intermediary-based middleware.

middleware implementations are optimised so as to reduce network bandwidth consumption and to mask the unreliability of mobile networks. However, use of mobile devices exclusively in hosting mobile services limits service scalability and availability—since there are no additional resources on the fixed network to draw on. Furthermore, the embedded middleware approaches do not solve the reachability problem; instead they rely on network-based solutions (discussed shortly). This limits their ability to host mobile services in practice and to support push notification.

With intermediary-based middleware (Figure 16), clients simply direct their service requests to the intermediary machines on the fixed network, using their well-known and static addresses. The middleware that is distributed across the intermediary and device offers the potential to provide a solution to the reachability problem. MobileHost [12] [13] [14], Nokia Mobile Web Server [5] and MSP (Mobile Service Platform) [15] [16] [17] take the intermediary approach.

Nokia Web Server adopts the intermediary architecture purely to address reachability for mobile services. It uses a HTTP-based protocol, where mobile devices periodically send GET requests to the intermediary. The intermediary queues service requests from clients and relays them to the device in the response message corresponding to the HTTP GET call. Mobile devices return responses to their service invocations as payload data of subsequent HTTP GET requests made of the intermediary. With MSP, the communication protocol to be used between mobile devices and the intermediary is undefined, and must be designed and implemented by application developers. Given that the communication issues are quite complex, implementing the equivalent of an Odin interconnect and communication layer is a non-trivial task that is better handled by middleware.

Intermediary-based approaches have the potential to address more than reachability. MSP, like Odin, allows for application components to be deployed on

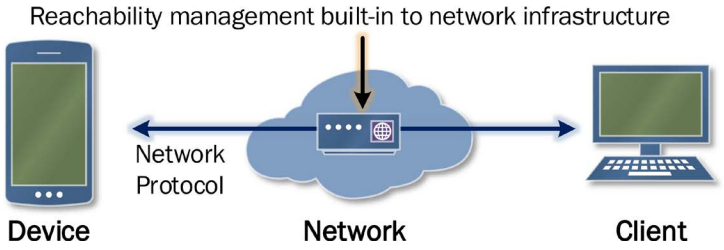

Figure 17. Network-based solution overview.

the intermediary to enhance the mobile application's scalability and availability. Odin further leverages the intermediary through its support for reconfiguration —surrogate migration and interconnect switching—and other services.

Other intermediary-based approaches such as MAUI [18] and CloneCloud [19] extend support for application partitioning, providing support for run-time decision making concerning whether a task should be executed by the device or on a machine on the fixed network. The middleware takes into account anticipated computation time and use of network resources in determining where a task should execute. However, MAUI and Clonecloud have not been designed with mobile services in mind, and so lack a solution to the reachability problem.

Network layer solutions (Figure 17) use the network infrastructure itself to address the reachability challenge. Examples include Mobile IP [20] and the GPRS tunneling protocol (GTP) [21] [22]. These solutions allocate devices with a static IP address that can be published to clients. Clients use this well known address to make service requests, and when the device's point of network attachment changes, the networking infrastructure is responsible for routing the request through to the new point of attachment. However, there is a lack of a single, widely-adopted mobile IP protocol at present, limiting developers' ability to rely on these solutions when developing mobile services in practice.

7. Conclusions and Future Work

In this paper, we have presented an overview of the Odin middleware that has been developed to facilitate development of networked mobile applications and services. The middleware addresses several challenges inherent in the mobile service domain through, fundamentally, a surrogate architecture. With the surrogate model, much service processing can be offloaded from the mobile device to a surrogate on the fixed network—enhancing a mobile application's scalability and effective availability. Furthermore, the surrogate architecture enables a solution to the reachability problem, allowing clients, via the surrogate, to initiate communication with mobile devices. Odin's interconnect for mobile networks solves the reachability problem that is generally presented when using mobile networks.

Odin is based on a component model. The middleware can be configured to use particular component implementations—notably interconnects—depending on application needs and operating environments. Furthermore, Odin can be

dynamically reconfigured, for example, to switch between different interconnects. The middleware ensures communication reliability during any interconnect reconfiguration. In addition, surrogate components can be migrated between surrogate hosts, again, without compromising application integrity. Other core components, including the communication layer and caching services, are substitutable and can be switched to suit particular application requirements.

Odin offers a technology agnostic approach to communication. Given that it supports the development of non-trivial networked mobile applications, Odin offers a rich set of communication primitives. This includes push notification, which is embedded in mainstream mobile operating systems. For push notification, Odin affords developers much freedom in the push behaviour of their applications. This is in contrast to use of native push notification APIs that impose restrictions on the way that they are used. Odin also offers a standards-compliant interface for clients to make push requests. As a result of Odin's design for remote communication, mobile applications can be developed with common communication behaviour over heterogeneous devices and communication links. Based on a comparative performance evaluation, Odin's messaging capabilities have been demonstrated to perform well with respect to industry push solutions.

Our ongoing work is aimed at increasing Odin's support for platform heterogeneity using *model-driven engineering* (MDE) techniques. We are developing a domain-specific modeling language for mobile-service applications whereby models are automatically transformed into platform-specific implementations. Initial results demonstrate feasibility of the approach and indicate that the tools can enhance productivity when developing networked mobile applications and services. We also intend to integrate cross-platform mobile development technologies, such as PhoneGap [23], since these industry tools have proven effective in developing cross-platform mobile user interfaces.

References

[1] Araujo, R., Igreja, A., de Castro, R. and Araujo, R.E. (2012) Driving Coach: A Smartphone Application to Evaluate Driving Efficient Patterns. *Intelligent Vehicles Symposium*, Madrid, 3-7 June 2012, 1005-1010. https://doi.org/10.1109/ivs.2012.6232304

[2] Fazeen, M., Gozick, B., Dantu, R., Bhukhiya, M. and Gonzales, M.C. (2012) Safe Driving Using Mobile Phones. *Intelligent Transportation Systems*, Anchorage, 16-19 September 2012, **13**, 1462-1468. https://doi.org/10.1109/tits.2012.2187640

[3] Krishna, S., Boren, S. and Balas, A. (2009) Healthcare via Cell Phones: A Systematic Review. *Telemedicine and E-health*, **15**, 231-240. https://doi.org/10.1089/tmj.2008.0099

[4] Meads, A. (2015) A Holistic Approach to Mobile Service Provisioning. Ph.D. thesis, Department of Computer Science, University of Auckland, Auckland.

[5] Wikman, J. and Dosa, F. (2006) Providing HTTP Access to Web Servers Running on Mobile Phones. Nokia Research Center, Espoo.

[6] Warren, I., Meads, A., Srirama, S., Weerasinghe, T. and Paniagua, C. (2014) Push

Notification Mechanisms for Pervasive Smartphone Applications. *IEEE Pervasive Computing*, Arlington, 24-28 March 2014, **13**, 61-71.

[7] Heger and Dominique A. (2012) Mobile Devices—An Introduction to the Android Operating Environment Design, Architecture, and Performance Implications. *DHTechnologies* (*DHT*), 43-49.

[8] Joseph, A.D., Tauber, J.A. and Kaashoek, M.F. (1997) Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, **46**, 337-352. https://doi.org/10.1109/12.580429

[9] ZeroC, Inc. (2009) Ice-E. ZeroC, Inc., Jupiter.

[10] Henning, M. and Spruiell, M. (2009) Distributed Programming with Ice. Revision 3.3.1b, ZeroC, Inc., Jupiter.

[11] Schmidt, H., Köhrer, A. and Hauck, F.J. (2008) SoapME: A Lightweight Java ME Web Service Container. *Proceedings of the 3rd Workshop on Middleware for Service Oriented Computing*, Leuven, 1-5 December 2008, 13-18. https://doi.org/10.1145/1462802.1462805

[12] Srirama, S.N., Jarke, M. and Prinz, W. (2006) Mobile Web Service Provisioning. *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, Guadelope, 19-25 February 2006, 120. https://doi.org/10.1109/AICT-ICIW.2006.215

[13] Srirama, S.N., Jarke, M. and Prinz, W. (2008) MWSMF: A Mediation Framework Realizing Scalable MobileWeb Service Provisioning. *International Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, Innsbruck, 13-15 February 2008, 1-7. https://doi.org/10.4108/icst.mobilware2008.2797

[14] Srirama, S.N. (2014) Mobile Web and Cloud Services. In: Bouguettaya, A., Sheng, Q.Z. and Daniel, F., Eds., *Advanced Web Services*, Springer, New York, 501-525. https://doi.org/10.1007/978-1-4614-7535-4_21

[15] Halteren, A.V. and Pawar, P. (2006) Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning. *IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, Montreal, 19-21 June 2006, 292-299. https://doi.org/10.1109/wimob.2006.1696373

[16] Pawar, P., van Beijnum, B., Peddemors, A. and van Halteren, A. (2007) Context-Aware Middleware Support for the Nomadic Mobile Services on Multi-Homed Handheld Mobile Devices. *Computers and Communications*, Las Vegas, 1-4 July 2007, 341-348.

[17] Pawar, P., Wac, K., van Beijnum, B., Maret, P., van Halteren, A. and Hermens, H. (2009) Context-Aware Middleware Architecture for Vertical Handover Support to Multi-homed Nomadic Mobile Services. *ACM Symposium on Applied Computing*, Bradford, 26-29 May 2009, 481-488.

[18] Cuervo, E., Balasubramanian, A., Cho, D.-K., Wolman, A., Saroiu, S., Chandra, R. and Bahl, P. (2010) Maui: Making Smartphones Last Longer with Code Offload. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys'10*, San Francisco, 15-18 June 2010, 49-62. https://doi.org/10.1145/1814433.1814441

[19] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M. and Patti, Ashwin. (2011) Clonecloud: Elastic Execution between Mobile Device and Cloud. *Proceedings of the Sixth Conference on Computer Systems, EuroSys'11*, Salzburg, 10-13 April 2011, 301-314. https://doi.org/10.1145/1966445.1966473

[20] Perkins, C.E. (1998) Mobile Networking Through Mobile IP. *IEEE Internet Com-*

puting, **2**, 58-69. https://doi.org/10.1109/4236.656077

[21] 3GPP (2009) 3GPP TS 29.060: General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp Interface. Technical Report v8.8.0. 3GPP, Technical Specification Group, Core Network and Terminals.

[22] Patil. B. (2003) IP Mobility Ensures Seamless Roaming. *Communication Systems Design*, **9**, 10-20.

[23] Wargo, J.M. (2012) PhoneGap Essentials: Building Cross-Platform Mobile Apps. Addison-Wesley, Boston.

Appendix

1.1. Push Notification OTA Protocol

HTTP is often used to realize the OTA link. Push content is delivered using the HTTP POST method, implying that the mobile device plays the server role. In addressing the reachability problem, the specification includes device-initiated connection, which relies on the PPG prompting the device to establish the connection with the PPG. This involves a session initiation request (SIR), that can be sent in a SMS message from the PPG to the device. The device then responds by initiating a connection with the PPG that the PPG uses to transmit push message(s). Since support for SMS is ubiquitous in mobile networks, with clients having well-known and accessible addresses (MSISDNs), the reachability problem is solved.

Other than HTTP, the OMA specification also recognizes that the OTA protocol can be layered on WSP and SIP. WSP (Wireless Session Protocol) is part of the WAP (Wireless Application Protocol) suite and essentially provides functionality similar to HTTP, but over lower layer WAP protocols that are optimized for use over wireless networks with relatively high data loss rates. WSP can be layered ultimately over a range of bearers that are commonly used by mobile network operators. These include IP-based protocols (GPRS and UMTS) and others (SMS and USSD). The specification also allows for connection-oriented and connectionless OTA variants, depending on the transport used. With a connection-oriented protocol, a logical connection is first set up between the PPG and device prior to exchanging push content. Following this the client can confirm receipt of the push notification using the connection. HTTP provides only connection-oriented push, while WSP and SIP support both connection and connectionless modes. The choice of mode should be informed by application requirements; connection-oriented protocols allow for delivery confirmation but at the additional overhead of setting up a connection.

The OTA protocol may alternatively be realized using a point-to-multipoint bearer. Such bearers include MBMS (Multimedia Broadcast/Multicast Service) and CBS (Cell Broadcast Service) and offer the ability to multicast or broadcast push content to groups. CBS, for example, allows the actual recipients of a push notification to be identified at run-time based on their geographical area. When using a point-to-multipoint bearer, a PI typically supplies a group name, as opposed to explicit device identities, and the group name is resolved to many devices by the bearer.

1.2. Push Notification PAP Protocol

The PAP governs interaction between the PPG and PI. HTTP is used as the underlying transport, with request messages taking the form of a HTTP POST. The PAP supports the following operations that are offered by the PPG: *submission*, *cancellation*, *replacement*, and *query*.

A push submission contains 3 elements: control, push content and device capability. The control part includes addressing data, comprising device address

(es) or a group identifier plus a URI to identify the device-hosted agent/application that should process the notification, any expiration time concerning delivery of the notification, and any QoS requirements. Capability data defines any assumptions that the PI makes of targeted device(s), such as their ability to process a given type of content.

In cases where a PPG stores a push message for later delivery, the PI can make a cancellation request. The PPG responds by removing the push message from its store and makes no further attempt to deliver it.

The PI can request that an earlier push request be replaced with a new one. For an earlier message with a single destination, assuming it has not already been delivered it will be canceled. Where the earlier message is associated with multiple destinations, the push replacement message can specify whether the replacement message should be sent to all recipients or only those that have not received the original message.

A PI can inquire as to the delivery status of a particular push request. The PPG responds with an indicator, such as the message is queued, has been delivered, or that delivery has failed. A PI can also query the PPG regarding capabilities of a particular device using this operation.

In addition, the PAP includes a *result notification* operation, directed towards the PI. This is an asynchronous operation that is invoked by the PPG to inform the PI about the delivery status of a particular push message. To register for the asynchronous update, PIs include an extra piece of control data—a notification URI—in a submission message. Once the outcome of handling the push submission is known, the PPG directs the result notification to the given URI.