



School of IT  
Technical Report



**The University of Sydney**

**WALKABOUT: AN ASYNCHRONOUS  
INTERNET MESSAGING ARCHITECTURE  
TAILORED TO MOBILE DEVICES  
TECHNICAL REPORT 579**

ADAM HUDSON AND BOB KUMMERFELD

JANUARY, 2006

# Walkabout: An Asynchronous Internet Messaging Architecture Tailored to Mobile Devices

Adam Hudson  
University of Sydney  
NSW Australia  
ahudson@it.usyd.edu.au

Bob Kummerfeld  
University of Sydney  
NSW Australia  
bob@it.usyd.edu.au

## Abstract

*Walkabout is an architecture that provides network support for message transfer between mobile devices communicating across the Internet. In order to cope with changing addresses and device disconnections, devices send messages to each other across a peer-to-peer overlay network, via peers which are located in the same local network segments as they are. This enables them to transfer messages at local network speeds, maximising their usage of transient network connectivity. The overlay uses a transfer protocol that makes use of parallel downloads wherever possible. Messages migrate to follow destination devices as they move, and are cached when the destination is not available. We have designed an approach that is robust and efficient and intend to prove this through simulation and a series of experiments.*

## 1 Introduction

Internet data transfers involving mobile devices can be difficult without dedicated system support. Yet, with the increasing ubiquity of devices such as digital cameras and portable media players, there is a growing need to be able to perform them.

Mobile devices are prolific producers and consumers of digital data. Digital cameras can hold hundreds of photos at a time, but these are prone to loss, for example if the storage media is corrupted or if the camera is stolen. An application running on a wirelessly enabled camera, which backs up this content automatically across the Internet, would be an ideal solution to this problem. Similarly, portable media devices can often hold in the order of gigabytes of data, but this is still likely to be only a subset of the total media library that the user owns or has access to. Their utility could be extended by allowing the user to request media not currently on the device from some other source, and have

it transferred as soon as possible. These applications would be possible today if it were feasible for a mobile device to carry out large data transfers across the Internet. However, there are many issues that complicate this.

Before a device can begin a transfer, it needs to obtain some form of network access, usually wirelessly. The ability to do so depends on the coverage available, and this can be highly variable. Once network access is established, a direct transfer between two hosts requires a constant, unbroken connection. Maintaining a connection again depends on the coverage, but low signal strength and the movement of the device out of range can affect the availability greatly. Local network speeds are often much faster than Internet connection speeds, so harnessing this difference could decrease the amount of time a mobile device needs to be connected to the network, minimising the impact of these problems.

### 1.1 Example application

As mentioned previously, a wireless camera that automatically backs up its contents across the Internet would be a valuable tool. It could work in practice if, as a person walked around a city, their camera turned itself on periodically to check for wireless networks.

Take the example of a couple on holidays. Throughout the course of a day, they take a hundred and fifty photos, each 2MB in size, on a wireless digital camera with a 256MB memory card. After each new photo, the camera turns on its wireless interface to check for network coverage. If it finds a network, the camera uploads the new image across the Internet to a backup server in their home. If the camera loses the network connection, or there isn't one available, it turns off the wireless again.

These people spend most of the day walking around, stopping for brief periods at locations of interest, for extended periods for lunch and for afternoon coffee, and pass numerous wireless hotspots of varying quality and upload speed along the way. While the camera has photos that need

to be backed up, it turns wireless (and itself, if necessary) on briefly every two minutes to check for coverage. When it finds one of these hotspots, the camera uploads as fast as the connection will support until it loses the connection, or all the photos are uploaded.

Towards the end of the day, the couple find that they run out of space on the memory card. However, they can see that most of the photos have been backed up, so they choose to delete all backed up photos. This frees up enough space on the card for them to not have to worry about it for the rest of the day. When they eventually return home at the end of their trip, all of their photos are waiting for them on their server, ready for sorting and reminiscing.

This example is quite feasible, as there are already wireless digital cameras on the market. Without effective network support, though, uploads may take too long to be practical, such that nobody would take advantage of the service. The architecture presented in this report can provide this support.

## 1.2 Walkabout

In this report we propose Walkabout, a messaging architecture designed to support asynchronous data transfers where at least one of the participants is likely to be a mobile device. It enables a device to send a message across the Internet to another device in a robust and efficient way that maximises the use of network connection time available to them. Messages are discrete and can range from small inter-application communications up to large media files many hundreds of megabytes in length. While they are delivered as quickly as possible, they should not be time critical, so Walkabout is not suited to interactive applications or streaming media, for example.

The system is implemented as a peer-to-peer store-and-forward overlay on top of IP, with peers located within the same local network segments as the communicating devices. Its key design features are:

- *Location independent addressing:* A sending device addresses a message to a destination device using a location independent key. The key resolves to the device's last known location via a distributed object location and routing (DOLR) service.
- *Decentralised delivery:* The overlay does not require any form of central storage or coordination to deliver messages.
- *Message segmentation:* A message is broken into pieces during transit, so that a transfer may be interrupted and resumed as necessary. Segmentation also allows multiple sources to provide different pieces of the same message simultaneously, potentially resulting in accelerated transfers.

- *Upload caching:* A device may upload a message to the Walkabout overlay as quickly as its local network allows. If the upload speed across the Internet is slower, then the message pieces are cached and transferred across the overlay at a later time, as fast as the connection allows.
- *Download caching:* The overlay transfers each message piece across the Internet to where the destination device was last seen. If the device is still present, the piece is delivered immediately. If the device is not present, the piece is cached until the device reconnects to the network.
- *Message migration:* If a device connects at one point in the overlay, but message pieces are being held for it at some other point, they migrate to the device's new location.

The overlay is formed from *proxy* nodes located in the local networks of people or organisations that use the service. A *client* running on a potentially mobile device creates a message addressed to one or more destination clients then uploads all of its pieces to a nearby proxy. This client is known as the *producer*. The proxy where each destination client was last seen downloads the message from the source, as well as from any other proxy with pieces of the message, using a store-and-forward file transfer protocol. These proxies deliver the message to the destination client devices, or *consumers*, whenever they are available, until they have the entire message.

Even with the rise of high-speed, 'always-on' mobile networking technologies, Walkabout still remains highly relevant. There is a wealth of devices already in use that are suited to make use of it, so it can be deployed without any need to upgrade networking hardware. Additionally, 'always-on' devices are still prone to disconnection due to lack of coverage, flat batteries and deliberate switching off, so the asynchronous capabilities we provide are still applicable.

This report makes two contributions. The main contribution is an architecture that allows mobile devices to transfer data across the Internet in a way that maximises their connection utilisation. In some cases, device mobility paired with downloads from multiple sources can facilitate a faster transfer rate than would be possible over a direct connection between static devices. The second contribution is a novel application of a peer-to-peer transfer protocol that tailors its rules to suit delivery to mobile devices.

The remainder of this report is organised as follows: Section 2 presents the relevant background material; Section 3 describes the key architectural components of Walkabout; Section 4 outlines the protocols and algorithms that govern the transfer of messages across the overlay; Section 5 outlines the methods that will be used to evaluate the system;

Section 6 outlines the differences between Walkabout and similar systems; Section 7 describes some future possibilities and Section 8 concludes the report.

## 2 Background

The work presented in this report draws upon existing research into techniques for the delivery of data to mobile devices across large networks, with an emphasis on message oriented middleware (MOM). Peer-to-peer file transfer protocols, particularly BitTorrent, are also explored.

### 2.1 Network support for direct communications

Traditionally, communications within a distributed system involve fixed hosts with stable network connections. The host initiating a transfer expects a correspondent to always be reachable at the same address, and the hosts at both ends must remain connected for the duration of the transfer. If hosts in this environment are able to move and change their address, they require specialised system support to continue these transfers.

Under dynamic DNS [17], a host is always referenced by the same domain name, which is bound to an IP address that the host updates whenever it changes. An application wishing to contact this host resolves the name and contacts its IP address. If a host moves during a transfer, the correspondent hosts needs to resolve the address again before it can continue.

Mobile IP (MIP) [11] hides physical mobility at the network layer, allowing a mobile host (MH) to always be reachable at a constant home IP address. When a MH moves beyond its home network, it alerts its home agent (HA) of its current globally accessible IP address. Any IP packets that a correspondent host sends to the MH's home address are intercepted by the HA and tunneled to the currently registered address instead. A MH may change its address without affecting the transfer.

The support that these protocols provide hides client mobility from applications running on the hosts, but both sender and receiver must be connected simultaneously for data delivery to succeed. This makes them a poor choice for communications between mobile devices, which may find such connectivity constraints difficult, due to prolonged periods of disconnection.

### 2.2 Asynchronous messaging

Asynchronous messaging solutions remove the tight coupling between hosts that use direct communication protocols, so that a producer may send data to some consumer

without needing to contact it directly. Instead, their store-and-forward nature allows a producer to place a message into the system without needing to know the addresses or even the identities of the consumers, and have it delivered to whoever is available. In some cases, the system stores the message until the consumer becomes available. The support for disconnected operation that asynchronous messaging provides makes it a suitable paradigm for communications between mobile hosts, which move frequently and may disconnect from the network for extended periods.

One form of asynchronous messaging has the producer storing its data at a rendezvous point for later retrieval by one or more consumers. A common example of this is email, where a message is addressed to an identity, delivered to a server, and stored until someone chooses to download it. A similar concept can be seen with tuple spaces, as demonstrated by Linda [8], where applications share data tuples by writing to and reading from a persistent, globally shared memory. Similarly, there are a variety of applications that take advantage of the strongly decentralised storage options provided by distributed hash tables (DHT) and distributed object and location (DOLR) frameworks, such as CAN [12], Kademia [10], Pastry [13] or Tapestry [20].

These systems provide a persistent store for the data and remove the need for either the producer or the storage point to have any knowledge of the recipient's current location. This makes them a robust, yet potentially inefficient choice for mobile communications. Each message needs to be transferred twice, to the store and then to the consumer. This is particularly inefficient if there is a lot of data and the producer and consumer are closer to each other than they are to the store. Also, these systems tend to require periodic polling on behalf of the consumer to check for new data. This can result in a consumer receiving a message some time after it was generated if the period is too long, or in excessive network traffic if the period is too short.

The publish/subscribe (or pub/sub) MOM paradigm presented by systems like Elvin [14] and Siena [4] also removes the need for the producer to know the addresses of a message's recipients. *Subscribers* register their interest in certain events (and their current location) via *subscription* to an *event broker*, which may be a single server or a network thereof. A *publisher* generates a message and delivers it to the event broker, which forwards it as a notification to any subscriber with a matching subscription.

A pub/sub system may be topic-based or content-based. In a topic-based system, a client subscribes to one or more topics, then receive a notification whenever a message is published to one of them. This is similar to joining a multicast group, as a publisher can easily send one message to many subscribers. Alternatively, subscribers in a content-based pub/sub system register subscriptions that contain expressions for the messages they would like to receive. When

the broker receives a new message from a publisher, it applies these expressions to the contents of the message, and delivers a notification to each matching subscriber.

The publish/subscribe paradigm does not intrinsically support disconnected operation, so if a subscriber is not connected to the service when a message is published, it does not receive it. However, there are several systems that allow subscriptions and events to persist even if the subscriber is absent.

Elvin supports mobility through the introduction of a proxy [16], which operates between a mobile client and the Elvin server. A proxy stays permanently connected to the Elvin server as if it were a regular client, and any subscriber wishing to use the Elvin service connects to it instead. The proxy subscribes to the server on behalf of its consumers and forwards any notifications it receives for them. Should the consumer be absent when a notification arrives, the proxy stores it until it reconnects. To reduce the number of notifications stored, a consumer may specify a time-to-live value or a maximum count for the results of each of their subscriptions. There is no support for the transfer of notifications between proxy servers, so the consumer must connect to the same one each time if it wishes to access stored notifications.

The Java Event-based Distributed Infrastructure (JEDI) [7] is a pub/sub architecture with native support for mobility. An active object (AO) registers a content-based subscription with the event dispatcher (ED), which can be deployed as a centralised server or a hierarchy of servers. When some AO sends an event to the ED, the ED notifies all other AOs with a matching subscription. If an AO wishes to disconnect from the network, it explicitly sends a `moveOut` request to the ED. This causes the ED to store the AOs subscriptions (so that it doesn't need to reregister them upon reconnection) and, optionally, any matching notifications generated during its absence. Upon reconnection, the AO sends a `moveIn` message and retrieves any stored messages. If the AO connects to a different ED server than the one it was previously at, the two servers communicate to deliver all stored messages to the AO. The correct behaviour of the mobility support depends upon an AO being able to explicitly sending a `moveOut`, so if it disconnects unexpectedly, none of its notification are stored.

The work of Caporuscio et al [3] presents a mobility support service that is similar to JEDI, but is independent from any specific underlying pub/sub service. They place mobility service proxies at various points throughout the network. While a subscriber is connected, it communicates directly with the pub/sub service. However, in the same manner as JEDI, the subscriber sends `moveOut` and `moveIn` messages to the proxy as it leaves and joins the network. On a `moveOut`, the proxy uses a customised module to subscribe

to the particular pub/sub service on the subscriber's behalf, and stores any notifications it receives. As with JEDI, this service allows a subscriber to send a `moveIn` to a different server than the one it sent the `moveOut` to, but it will also not operate correctly if the subscriber disconnects unexpectedly.

## 2.3 Java message service (JMS)

The Java Message Service (JMS) [15] is a messaging API that supports both point-to-point and publish/subscribe message delivery models. JMS clients, which may be on the same or on different devices, exchange messages through a JMS provider. Delivery under either model may be synchronous, where a client blocks until it receives a message, or asynchronous. In the point-to-point model, consumers register with a queue on the provider, a producer delivers a message to the queue and the provider forwards to at most one of the consumers. If there are no consumers available, the provider attempts to hold the message until one connects. A queue can be configured for persistent messages, such that the provider stores them to disk for guaranteed delivery. JMS also supports topic-based publish/subscribe messaging, with durable subscriptions providing a mechanism for disconnected subscribers to receive their messages upon reconnection. Both queues and topics require an administrator to create them.

Being an API, JMS doesn't have any specific optimisations for mobile devices, beyond the requirement of support for persistent messages and durable subscriptions. The properties of the system depend upon the implementation, and `iBus//mobile` and `Pronto` are two that are specifically tailored to delivery to mobile devices across a large network, such as the Internet.

`iBus//mobile` [9] provides mobile devices with a gateway, which acts as a proxy for communication with a JMS provider running on a central server. The gateway offers transcoding functions for Java enabled devices, so that the data they receive is better suited to their limited capabilities. It also provides an interface between JMS and other technologies such as email, SMS and WAP. There is no discussion of how the placement of multiple gateways could improve performance for mobile devices, and the architecture is server based.

`Pronto` [19] is another JMS implementation that has the option of gateway. It performs similar transcoding and interface functions to those seen in `iBus//mobile`, but adds "SmartCaching" [18] to improve the mobile experience. This allow a device to synchronously retrieve a single message from the cache, to asynchronously receive updates as they arrive, or to obtain a snapshot of any new data since the last request. `Pronto` also provides a serverless version of its mobile client, which uses IP multicast to run JMS in

a decentralised fashion. However, in doing so it omits the point-to-point transfer model and durable subscriptions, and the lack of wide-scale support for IP multicast would make it difficult to scale this version to the Internet.

## 2.4 Peer-to-peer file transfers

Peer-to-peer file transfer protocols such as BitTorrent and Gnutella<sup>1</sup> or those behind the applications KaZaA<sup>2</sup> and eDonkey<sup>3</sup> provide a powerful mechanism for the cooperative downloading of files, particularly larger ones, across the Internet. A single web site may be unwilling (or unable) to serve a large file to hundreds of users simultaneously. Peer-to-peer protocols leverage a host's often underutilised upload capacity to serve any parts of the file that they have to each other. This results in clients downloading from multiple locations at once, often at a higher overall transfer rate than would be possible over a single connection to a web server.

BitTorrent [5] is a protocol where an overlay network is formed dynamically from peers with a mutual interest in a given file. The overlay is initially created by the deployment of a tracker, a .torrent file, and a seed. The tracker is a program running on a web server, which manages the overlay by informing hosts of the IP address and listening port of others that are also downloading the file. A .torrent file contains information about the file being transferred, including its name, length, the tracker address, and hash signatures of each of the pieces that make up the file. This is placed on a web site for download. The seed is a complete copy of the file that must be uploaded in its entirety at least once for others to obtain it.

To begin a download, a user retrieves a .torrent file and opens it with an application. The application contacts the tracker specified in the file to obtain a list of available peers (which initially contains just the seed) and registers its host as a new peer. The host then contacts each of its peers to find which pieces of the file each one can offer, selects smaller sub-pieces (known as blocks) of the ones available according to rules which aim to increase the system-wide availability of the file, and begins requesting several of them at once. Whenever it finishes downloading a block, the host requests a new one. An extra component of BitTorrent is the choking algorithm, which aims to improve total overlay performance by ensuring a peer will only upload data to those peers which are willing to upload themselves.

Each time a peer completes a full piece of the file, it verifies its hash against that provided in the .torrent file and, if they match, alerts each of its connected peers. Transfers continue until the host has the whole file, at which point it

<sup>1</sup><http://www.gnutella.com/>

<sup>2</sup><http://www.kazaa.com>

<sup>3</sup><http://www.edonkey2000.com/>

stays connected to the overlay as long possible to continue uploading as another seed.

The tracker is a central point of failure in the basic BitTorrent protocol, so modifications [6] make use of a Kademlia DHT [10] to store peer information. Portions of the header are hashed to form what is known as the info hash, which is used as the key to store the list of peers within the DHT.

A BitTorrent overlay becomes more robust as more peers join, but it does not have any kind of support for mobile devices. For example, if there are a small number of peers and they are all mobile, frequent disconnections may reduce the availability of the file and make it difficult for any of the non-seeds to obtain a complete copy.

## 3 Walkabout components

The two main components of Walkabout are the client and the proxy, and these are supported by a DOLR service and a service registry. This section explains the key features of these components.

### 3.1 Client

A client is an application that runs on a device and coordinates the transfer of messages to and from the overlay via its local proxy. Depending on the capabilities of the device, which can be mobile or fixed, and the task being performed the client may either perform its operations automatically or require user interaction to trigger them.

Each client has an RSA key pair which it uses as a basis for identification and cryptography. In particular, the SHA-1 hash of the client's public key (or *key hash*) serves as a globally unique identifier. The method used to distribute and manage these keys is outside the scope of this report.

Multiple clients can run on a device simultaneously, provided that each has its own key and is listening on a unique TCP port.

### 3.2 Proxy

A proxy is a member of the peer-to-peer overlay that provides client connection, caching and message transfer functions. One would typically exist on a single dedicated machine within a private network, similar to a web proxy. It could be provided as a free service to trusted users in a home or office environment, as additional value to accompany other services, e.g., in a café, or as a wide scale subscription service, deployed alongside wireless access points.

A proxy caches message pieces that it receives from clients and other proxies, until it is either able to deliver them to a proxy closer to the destination, or the destination itself. When the cache reaches its limit, the proxy deletes

pieces according to the cache purging policy, which can be selected to suit the proxy's capabilities. Pieces are written to disk, so if the proxy shuts down for any reason, it retains its cache when it restarts.

A single machine can run both a client and a proxy, though this is only suitable for fixed systems that are unlikely to disconnect, such as desktop computers. This dual operation can be used to provide bridged Walkabout access for devices with Bluetooth or USB, but lacking 802.11 or Ethernet interfaces.

### 3.3 DOLR service

Walkabout requires a scaleable mechanism for proxies to locate clients and message sources, and this is something that DOLR can provide. Therefore every proxy joins a DOLR overlay as they launch.

Using this overlay, proxies publish a *client location* entry whenever a client connects to them. This associates the client's key hash with the proxy's own IP address and TCP port, and allows other proxies to send data to the client's last point of attachment by simply providing the key hash. Similarly, *message tracker* entries associate a message's signature with the address of the proxy that is currently the coordinator of information about it. This information includes which proxies currently have pieces of the message, the key hash of each consumer that has completely received the message, and the key hash of each consumer it is addressed to that has yet to receive it all. Any proxy wishing to request or update this information contacts the message tracker by routing it to the message signature.

### 3.4 Service registry

Upon connection to a new network, a client needs some way to automatically find its local proxy. Additionally, if there are not any proxies available locally, then they should be able to find one in a remote network. Apple's zero configuration service discovery protocol, Bonjour<sup>4</sup>, can provide this functionality.

Walkabout therefore requires each proxy to run an mDNS responder, and the placement of a Bonjour service registry at a globally accessible URL known to the client. Each proxy has the option of adding itself to the Bonjour registry when it starts, if its administrator is willing to open the service to remote connections. This registry is not required for the system to function, but it does make it more robust, by allowing a client to use Walkabout even if there is no local proxy available.

---

<sup>4</sup><http://www.apple.com/macosx/features/bonjour/>

## 4 System design

This section explains how the components join together to form the Walkabout network and details the algorithms and protocols governing its operation.

### 4.1 Network formation

The proxies that form the overlay network at the core of Walkabout do not need to exchange much initial state information with their peers when they first connect. Rather, they communicate as required, when searching for clients and transferring messages.

Before they can participate in the overlay, though, they do need to register with the DOLR service and, optionally, the Bonjour registry. So when a proxy starts, it first boots into the DOLR overlay. This allows it to access entries when it needs to, while also acting as a potential storage point itself. If there are any undelivered messages remaining in its cache, the proxy requests the message tracker message via DOLR and then contacts any peers to inform them of the pieces it has available. If so configured, it also adds itself as a Walkabout proxy with a well-known Bonjour registry.

### 4.2 Protocols

A complete transfer requires several steps. First, the producer connects to the overlay and uploads a message. The overlay transfers this message to the appropriate proxies, and the consumer downloads the message from them. If new message pieces arrive at a proxy, but its cache is full, it purges existing pieces to make space. The following section explains how each of these steps work.

#### 4.2.1 Client connection

A client needs to find a proxy before it can access the system, so it searches the local network by sending out a Bonjour request. If it receives a response, it connects. However, if there does not appear to be a local proxy, it instead queries the service registry to find one in a remote network. While it is sub-optimal to use a remote proxy, as this restricts transfers to the speed of the Internet connection rather than the local network, it is unlikely that a Walkabout will be available in every network. This may therefore be the only option available at times. If it is still unable to find a proxy after querying the registry, due to firewall constraints for example, then the client is unable to connect to Walkabout.

When it connects to a proxy, the client initially provides its key hash, its IP address and listening port, and the address and port of the proxy it was last connected to. The proxy stores the client's address and port in a table indexed

by the key hash, then returns its own address details to the client and publishes them to the DOLR service under the key hash.

After the initial client registration, the new proxy contacts the client's previous proxy to inform it of the move. If it has any undelivered messages addressed for the client, the previous proxy sends across the message headers. The message download protocol is then initiated if required, with the previous proxy as the new one's initial peer.

#### 4.2.2 Client upload

To send a new message, a client must first specify the key hashes of the clients it wishes to deliver it to. Depending upon the application, the keys could be pre-configured (e.g., for a backup server) or selected from a list obtained through out-of-band means (e.g., from friends over email).

The client uploads the message in pieces to its proxy, accompanied by a header containing content and address details.

The content details are similar to what is contained within a BitTorrent .torrent file. They include:

- The total length (in bytes) of the message.
- The length (in bytes) of each piece of the message. The default length is 256KB.
- The SHA-1 hash of each piece.
- Application specific metadata, detailing such properties as the name of a file or the type of an inter-application message.

The SHA-1 hash of these fields, known as the *message signature*, uniquely identifies the message within the overlay. Because the entire message needs to be known to calculate the header, all transfers must be discrete messages, and therefore Walkabout does not support streaming data natively.

Additionally, the address details are:

- The key hash of the producer.
- A list of the key hashes of each intended consumer.

Upon receiving the header, the proxy uses the information it contains to contact its relevant peers. First, it creates a message tracker update with itself as the only peer and attempts to send it over the DOLR layer, using the message signature as the destination. If the delivery is unsuccessful, the proxy creates the tracker locally and then publishes its location under the message signature. However, if the delivery is successful, the proxy that is currently responsible for the tracker receives the update, merges any new values into the existing state, then returns the tracker to the requesting

proxy. Note that if the proxy that houses a tracker goes offline, the next proxy that attempts to contact the tracker will find that it is no longer available, create the tracker with the information that it already has, and publish itself as the new location.

Next, the proxy needs to locate and notify the consumers that have yet to receive the message. Using the information from the tracker, the proxy delivers the header via DOLR to the hash key of each appropriate client. The proxies that receive the header may then choose to initiate transfer of the message.

After uploading the header, the client starts transferring the actual message. It sends the message in pieces, each one of the size specified in the content details. The default length is 256KB, but this can be changed on a per-message or per-client basis. For example, a device may use smaller pieces if it has slower networking capabilities. When a piece upload is complete, the proxy verifies it against its hash in the header, and acknowledges it if it matches. The client marks it as uploaded. This continues until the entire message is uploaded, or the client disconnects. If a transfer is interrupted, a client sends the header again upon reconnection and then resumes the message transfer from the first unacknowledged piece.

When the client believes it has completed an upload, it checks with its proxy. If the proxy can see that all of the pieces of the message are available within the overlay, it informs the client that its upload is indeed complete, and the client knows that it can stop trying to upload it. However, if there are any pieces that cannot be found, e.g., if the proxy that held them disconnected unexpectedly, the proxy informs the client of which ones (by sending it a piece map) and it uploads them to complete the transfer. The concept of the piece map and the mechanism for discovering availability within the overlay are explained in the next section.

A client often wants to know when a message has been delivered to its destinations successfully. For example, a backup application needs to know when a file is safely in the archive so that the local copy can be deleted. Walkabout achieves this by having the proxy at the receiving end send a message back to the producer when it completes a transfer to the consumer. When the client receives notifications from all of the message destinations, the message has been completely delivered, and the application can act accordingly.

#### 4.2.3 Transfers

Message transfers across the Walkabout overlay are inspired by the BitTorrent protocol. They are driven by the proxies that are receiving on behalf of the consumers and make use of parallelism where possible. Proxies seeking the same message use a tracker to find each other, and can upload to

and download from multiple peers simultaneously.

Message states are communicated between nodes by way of *piece maps*. These are simply strings containing as many bits as there are pieces in the message. Depending on the context, the value of the bits have different meanings. If the map was generated by client downloading a message, a value of 1 indicates that the client wants the piece in the position corresponding to that bit, and 0 indicates that they don't. Similarly, the value of the bit in a map generated by a proxy indicates whether they have the piece or not. These maps are generally quite small, e.g., a one gigabyte file transmitted as 256KB pieces generates a map of only 512 bytes.

The transfer negotiation begins when one proxy receives a message header from another. This could be the result of a producer actively trying to send the message to a client registered at the proxy's location, or of a message update from the previous proxy for a client that has just connected locally.

Before the proxy can begin downloading a message, it needs to determine which pieces, if any, the clients it is responsible for want. If the proxy already has any of the clients' piece maps, it is able to start requesting pieces immediately. If not, the proxy tries to contact each client it requires a map from by forwarding the message's header. Until it receives a response containing the client's piece map, or if it never does (most likely due to client disconnection) it assumes that the client wants the entire message. If a response is received later, the proxy updates its stored map for that client and cancels transfer requests for any pieces that they do not want. A client may also choose to reject the transfer completely, which stops the proxy from asking it about that message again.

While it is consulting the clients, the proxy is also building a list of peers that may be able to provide the required pieces. The proxy that sent the header is automatically added as the first member of the list. Any others are located by the proxy consulting the message tracker.

Transfers are typically only to a small number of recipients, so the list of peers tends to be small enough that the proxy is able to contact all of them. If the list is larger than a locally configured threshold, perhaps ten or twenty, then it selects a random set.

The proxy queries each of its peers as it connects to them, to find what their piece map says they can offer for download. Even if a new peer has no required pieces, the connection is kept open because it may receive some in the future. For example, a producer may have connected to it in order to transfer a new message, but not started uploading pieces yet.

A Walkabout proxy requests data in blocks. Pieces are sub-divided into blocks  $\frac{1}{16}$  the size of a piece, making them each 16KB by default. Several blocks, typically five, are

requested simultaneously from each peer. As blocks arrive, new ones are requested. Once a proxy collects a full piece worth of blocks, it verifies it using the relevant hash value from the header. If the verification is successful, it marks the piece as downloaded in its own piece map, any pending downloads for blocks from that piece are cancelled, and the piece is delivered to clients as outlined in the next section. It also sends a short notification to each of its peers, which causes them to update their piece map for this proxy. If, however, the verification fails, possibly due to a transmission error or a malicious peer, the entire piece needs to be downloaded again.

The heuristics that a proxy uses to select which pieces, and therefore which blocks, to request from which peers have a large bearing on performance. The ones presented here are again based upon those used in BitTorrent, but are augmented by rules that aim to satisfy client needs as quickly as possible. They are applied in the following order:

- *Client satisfaction first:* The only reason for a proxy to download a piece is if it knows that a client it is currently responsible for requires it. So any block it requests must be from a piece that at least one of them needs.
- *Maintaining availability:* If a proxy starts running out of cache space, it needs to purge message pieces. The full mechanism behind this is explained later (see Section 4.2.5), but if a proxy receives notification of another's proxy's intention to delete, and it still needs to download the piece, it prioritises it over any others.
- *Client maximum satisfaction:* If the proxy is requesting a piece on behalf of multiple clients, it selects the ones that are needed by the most clients.
- *Rarest first:* Out of the pieces selected by the previous rules, the pieces that the least number of peers have are the ones most likely to become difficult to obtain in the future.
- *Strict priority:* Once a block has been requested, the remaining blocks from the same piece are requested before any others.
- *Random:* If, after applying the above rules, there are multiple potential pieces, the proxy selects one at random. This helps to increase the entropy of which pieces are available across the overlay, and should ultimately help contribute to increased transfer speeds through parallel downloads.
- *Endgame mode:* Once a proxy is actively requesting all the remaining blocks it needs, it issues additional requests for each of them to the other peers that have

them. This is to nullify the effect a slow peer could have in delaying the completion of the transfer. As each block arrives, request cancellations are sent out to each of the other peers the block was requested from.

An important aspect of the peer-to-peer transfer model that Walkabout uses is that the proxy and the Internet connection it uses are often shared between multiple users. As a result of the resources (download capacity, upload capacity, and network connections) that each message transfer requires, only a small number can be active at any given time. This can lead to there being more messages than there are available resources to upload or download them, so transfers need to be queued as they arrive, and scheduled in an appropriate manner.

The scheduling algorithm that is suitable for a proxy depends heavily upon the characteristics of the messages it handles and the number of users it services. The following are some examples of the forms of scheduling that may be applicable:

- *First come, first served:* For a single user proxy in the home, where there are not many transfers. The proxy performs transfers in the order they arrive.
- *Taking turns:* In an office with many users, where everybody wants equal access to the service. Messages are processed in the order that they arrive, but if a user has multiple messages in the queue at once, one transfer is scheduled for every other user before their next one is.
- *Some large, some small:* In any environment with many transfers. If there is a mix of large and small messages, the transfer slots are divided up between the two, where some threshold value defines what type a message is. This ensures that large messages do not unfairly delay the transfer of smaller ones.

Of the active transfers, some will be downloading and uploading simultaneously, while the remainder will be uploading to peers only. The type of transfer queue, the number of simultaneous downloads, the total number of simultaneous transfer and the various other transfer parameters need to be configured by an administrator to suit the characteristics of the proxy. The community that supports the BitTorrent program Azureus provides recommended values for the various settings, based upon real-world observations<sup>5</sup> of hosts with different upload speeds. Based upon these, it seems that between two and five active transfers is a suitable number, with about half of these allowing downloads.

<sup>5</sup>[http://azureus.aelitis.com/wiki/index.php/Good\\_settings](http://azureus.aelitis.com/wiki/index.php/Good_settings)

#### 4.2.4 Client download

The final phase in the delivery of a message is when the client downloads from its proxy. This could be triggered by notification of a new message arriving at the proxy, or by the client finding messages waiting for it upon reconnection.

If a client is connected when a new message arrives, the proxy consults with the client as explained in the previous section, then relays each piece as it receives them. The client verifies the piece against the hash in the header and acknowledges its receipt to the proxy if it is valid.

The other situation that triggers a download is when a client connects to a proxy. The client begins by registering, then uploading the signature and piece map for each incomplete message it wishes to continue downloading. This immediately gives the proxy an awareness of which messages and pieces the client needs. If it is the same proxy that the client was last connected to, this refreshes its state, and the proxy immediately starts sending any pieces to the client that it acquired in its absence. If it is a different proxy, though, this update allows it to make informed piece selections as message updates, triggered by the client's movement, start to arrive. As the proxy requests and receives new pieces, they are delivered to and acknowledged by the client in the manner outlined above.

Once the client returns the final acknowledgement, the message download is complete. The client can either finish writing the file to disk or process the application message, depending on the type of application. The proxy sends a completion announcement to each of its peers, and a *delivery acknowledgment* to the producer of the message. This delivery acknowledgment contains:

- The signature of the message being acknowledged.
- The key hash of the consumer.
- The key hash of the producer.

A delivery acknowledgment is a special message that is routed in a simpler manner than regular messages, because it doesn't have any payload. The proxy sends the acknowledgement directly to the producers last known proxy via DOLR. If the client is connected, the proxy delivers the message the same way as it would a header. If it isn't, the proxy caches the message to either deliver to the proxy when it reconnects, or to forward if it connects to a different proxy.

If a consumer indicates that it doesn't wish to download a particular message, this generates a *rejection notification* at the proxy that is asking it. This contains the same fields as a delivery acknowledgment, and is delivered to the producer in the same fashion.

After delivering the final piece of the message that a consumer needs, a proxy alerts each of its peers that this client

has received the message, and updates the message tracker. If there are no remaining consumers waiting for the message, each of the peers can safely remove themselves from the message tracker and delete any pieces of the message that they hold. The proxy that houses it may also safely delete the tracker at this stage.

#### 4.2.5 Purging

While the operation of Walkabout is dependent upon the use of caches, a proxy is limited in the amount of storage space that it can offer. As new message pieces arrive, existing ones may need to be removed, possibly even before they have been delivered to their destination clients. This could potentially prevent a complete message from ever reaching its destination, so intelligent purging rules are required to minimise the impact on the system of piece deletion.

First, a proxy looks at message pieces that are no longer required locally, either because they have been delivered to all local consumers, or because any consumers that wanted them have connected elsewhere. Any pieces known to be held on a peer are preferred for deletion, starting with the ones with the highest number of replicas available. Whenever a proxy deletes a piece, it sends a deletion notification to its peers, in the same way as a piece download notification, so that they can update their piece map.

If the proxy still needs to delete pieces, it attempts to purge some that it has no further use for, but that do not exist on any of the other peers. It asks the peers to take these pieces by sending them an *intention to delete* notification, leading them to prioritise the retrieval of these pieces over all others. The proxy holds the pieces until it receives a piece download notification from a peer. If it does not receive a response within an acceptable time frame, the proxy must instead try to delete something else, because deleting these pieces could result in them being lost permanently.

So, the next choice for the proxy is to start deleting the pieces it still needs to deliver. This is obviously undesirable, because it is likely that the proxy will need to retrieve it again at some point in the future. As above, it first tries deleting pieces that are replicated elsewhere, then those that are not.

In rare cases, a proxy may find that it is only able to delete message pieces that some of its clients still need, that don't exist elsewhere, and that no peers will download from it. In this situation, it starts with the pieces that the least number of consumers need, then the pieces that have been in the cache the longest.

## 5 Evaluation

This section briefly explains the effects that mobility has on transfers under Walkabout and presents our plan for eval-

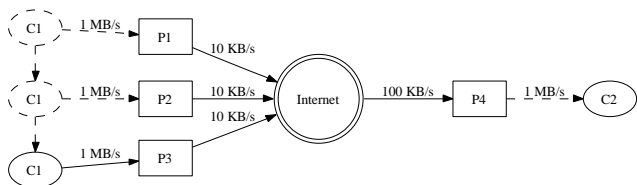
uation through simulation and implementation.

### 5.1 The effects of mobility

As a client moves around the network, or when a message is sent to multiple destinations, the number of peers for a message increases. This in turn increases the potential for parallel downloads, which can lead to improvements in the end-to-end transfer rate of a message and a decrease in the amount of time a device needs to spend communicating with a proxy to complete its transfer. Also, a consumer that disconnects but then reconnects to the same proxy will often experience greatly reduced local download times. In lieu of simulation results, this section presents some informal example scenarios that demonstrate how the mobility of a Walkabout client can lead to performance gains.

#### 5.1.1 Moving producer

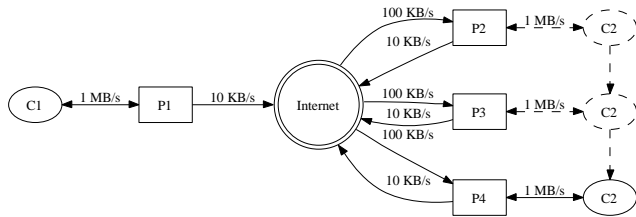
As a producer moves, it may find itself uploading different pieces of the same message to multiple proxies. When the message is large and the local network speeds are considerably faster than the available Internet upload speed per transfer, this movement leads to the proxy near the consumer having multiple points to download from simultaneously. These parallel downloads can result in a faster overall transfer, possibly even faster than would be possible in a direct transfer between the two clients.



**Figure 1. A producer moving between different proxies**

Take the example in figure 1. At each of the networks the wireless client  $C_1$  connects to, it is able to upload a message at 1MB/s to the proxy, which in turn has 10KB/s available upload bandwidth to the Internet.  $C_1$  wishes to send a 20MB file to the client  $C_2$ , running on a fixed backup server. It connects to  $P_1$  for ten seconds, disconnects for a minute, connects to  $P_2$  for five seconds, disconnects for another minute, then finally connects to  $P_3$  until the upload completes.

The proxy near the consumer,  $P_4$ , can download from the Internet at 100KB/s. If it begins downloading the file as soon as the notification arrives, it will initially get a speed of 10KB/s, then 20KB/s as  $C_1$  reaches  $P_2$  and 30KB/s as  $C_1$  reaches  $P_3$ . This is due to each proxy having received



**Figure 2. A consumer moving between different proxies**

unique pieces from the producer, which  $P_4$  can download in parallel. The speed will taper off again as  $P_4$  exhausts the pieces each proxy has to offer.

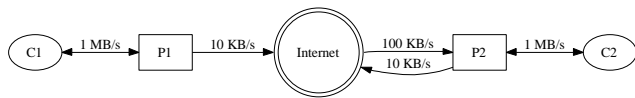
There are two important results to note here. Firstly, and most importantly when supporting mobility, the producer  $C_1$  only needed to hold a network connection for a total of twenty seconds to upload the entire file. By contrast, a direct transfer between  $C_1$  and  $C_2$  without the support of Walkabout would take in excess of thirty minutes.

The other result is that the average overall transfer speed is faster than would be possible over a direct connection between  $C_1$  and  $C_2$ . The actual speed depends on several factors, such as the time taken for a proxy to locate the consumer, the data overheads of the transfer protocol and link latencies. However, if we ignore these factors for the moment, a theoretical maximum of approximately 20KB/s is possible. This could be increased further if the producer visited even more distinct proxies. In practice, the speed would be lower than this theoretical value, but we can see that it would still be considerably greater than would be possible without the support of Walkabout.

### 5.1.2 Moving consumer

As with the previous example, a moving consumer can also lead to multiple proxies accumulating pieces of the same message, which another proxy can download from in parallel to accelerate their download. Figure 2 presents a fixed producer,  $C_1$ , sending a 5MB message to a moving consumer,  $C_2$ . In this example,  $C_2$  downloads while it is connected to  $P_2$ , but disconnects after a minute. During its absence,  $P_2$  continues to download, so when  $C_2$  reconnects at  $P_3$  thirty seconds later, there are now two sources for the message (i.e., at  $P_1$  and  $P_2$ ). As a result, the download to  $C_2$  via  $P_3$  experiences an initial burst of speed, until the pieces cached at  $P_2$  are exhausted.  $C_2$  leaves after another minute, then connects to  $P_4$  thirty seconds later, where it receives another initial burst of speed, and stays connected until it completes the message download.

If we again take a simplified approach, the average overall speed for this transfer is 10KB/s. Even though the con-



**Figure 3. A consumer disconnecting from and reconnecting to the same proxy**

sumer was disconnected for a minute, the speed is no different than if they had remained stationary. However, this added speed is at the expense of overall data traffic, as there were pieces that  $P_3$  downloaded from  $P_2$  and that  $P_4$  downloaded from  $P_3$  that were originally retrieved from  $P_1$ . If  $P_1$  had a faster upload speed, the gain in speed would be less pronounced, and if it were greater than any of the other proxy's download speeds, it would saturate the link such that Walkabout parallel downloading would not offer any performance gain.

From the perspective of the consumer device, the scenario in figure 2 results in only a small benefit. The end-to-end delivery was faster than if the client had been downloading from the producer directly when it was connected, but it still took approximately nine minutes to deliver, and the consumer had to be present for eight of them. Because the message is cached at the proxy, though,  $C_2$  could have disconnected from  $P_4$  for several minutes, then reconnected to download at local network speeds, with no change to the end-to-end delivery time.

Figure 3 explores the performance increase when a Walkabout client reconnects to the same proxy it was previously connected to. In this example, the consumer  $C_2$  connects to  $P_2$  for thirty seconds, then disconnects for an hour while the user is at lunch, during which time it makes no contact with any other Walkabout proxy. While  $C_2$  is absent,  $C_1$  generates the same 5MB file presented in the previous scenario and delivers it to  $P_2$ , destined for  $C_2$ . When the consumer reconnects, it takes only 5 seconds to retrieve the file at 1MB/s. This demonstrates how effective Walkabout can be at minimising the connection time required for a mobile device to download a message.

## 5.2 Simulation

A framework for system simulation is currently under construction using the OMNeT++ discrete event simulation environment<sup>6</sup>. The performance of Walkabout will be compared to standard techniques for the transfer of data in a mobile environment under a variety of different conditions. We will vary mobility patterns, topologies and message properties.

Burcea et al [2] suggest three different mobility scenarios that are suitable for our evaluation:

<sup>6</sup><http://www.omnetpp.org>

- *Commute*: Depicts a client that alternates between the same two connection points, in the home and in the office. It connects to the proxy in the office between 8am and 9am, spends about 8 hours in the office, disconnects for an extended period then connects to the proxy in the home.
- *Pervasive*: Depicts a roaming client that moves regularly between proxies, but attempts to maintain a connection whenever possible. It experiences short but frequent disconnections.
- *Random*: Depicts a client that alternates between periods of connection and disconnection for periods varying randomly between 10 and 30 minutes. While disconnected, they move at a random speed until they reconnect.

By varying simulation properties, we can obtain different overlay topologies and data traffic patterns. The properties will include:

- Link bandwidth and latency, for local networks, connections to ISPs and across the Internet.
- Number of local networks, and how many of them contain a proxy.
- Number and location of producers.
- Producer message creation rate
- Number and location of consumers.
- Message size

The same combinations of scenario and parameters that we decide upon will be applied to the communication models of direct transfers using mobile IP, indirect transfers via a pub/sub server and transfers using Walkabout. We will compare their performance in terms of end-to-end message transfer times, client transfer times (when connected), message overheads per MB of data, and average throughput. We expect that as the rate of client movement increases, Walkabout should outperform the other technologies in metrics relating to speed, though possibly at the expense of overall data traffic.

We will measure the impact of a proxy performing multiple simultaneous transfers upon performance, and the efficiency of sending a message to multiple destinations.

We also intend to test how modifying the piece selection and purging rules will affect overall system performance, and whether a BitTorrent choking mechanism is required.

## 5.3 Implementation

To prove that Walkabout works in a practical context, we plan to produce a working development library. This will allow us to create two distinct applications running on mobile devices, to gain some real world measures of performance.

### 5.3.1 Audio and video messaging

The Keep-in-Touch messaging system (KIT) [1] provides a means for asynchronous communication through messaging appliances. It allows people who are disconnected in time and space to communicate via audio and video messaging. All of the devices KIT runs on currently are fixed, but Walkabout would be the ideal architecture to enable the inclusion of mobile clients.

Therefore we aim to produce an implementation of KIT running on an audio and/or video enabled mobile device, most likely a PDA. We will run this application through a series of experiments, where we create and receive messages while moving, to measure how effective Walkabout is in practice at transferring application messages.

### 5.3.2 Automatic photo backup

The ideal appliance that this technology could enable is the wifi equipped camera presented in the introduction to this report. A Walkabout client running on the camera could automatically control the backup operations.

The client is configured with the address of a backup service, which is another client that may be running on a machine they own or possibly as a commercial service in a data storage facility.

While there are new images on the camera, the client activates the wifi interface at regular, user-configurable intervals to check for wireless network coverage. If the client makes contact with a proxy, it uploads each new image as fast as the local connection allows, as a message addressed to the backup service. As each image upload completes, the image is marked as 'Uploaded'.

When it is connected, the client may also receive delivery acknowledgments for previously uploaded images. The acknowledged images are marked as 'Backed up' on the camera. The owner then knows that they may delete the images safely if they need space for more photos, either individually or through a function to delete all backed up images.

If the camera doesn't complete all the transfers, it goes back to checking periodically for network coverage. If it does manage to upload all of its images, it turns off the wireless until there are new ones. However, a user-launched function allows them turn on the wireless to check for acknowledgements if they wish.

A user-programmable piece of hardware that would allow us to deploy this program is unlikely to become avail-

able any time soon, so we plan to emulate it with either a Bluetooth enabled camera phone or a wifi enabled PDA. As with the KIT example, we will run a series of experiments to measure Walkabout's performance when transferring files.

## 6 Comparison to existing systems

The message delivery model of Walkabout is similar to that of topic-based pub/sub. In traditional pub/sub, the decoupling of sender and receiver allows a publisher to create messages that any number of subscribers can receive, without the publisher ever needing to know anything about them. We don't want this abstraction, though, because even though the producer doesn't know the address of a message's consumer, it does know their identities, and wants to make sure they are who the message is delivered to. So Walkabout's delivery model is different, because even though a consumer registering their location against their key is similar to a subscription, the producer explicitly selects each recipient individually by specifying their key.

Message persistence in JEDI [7] is supported by proxy-like servers that are responsible for storing messages during a mobile client's absence, and migrating them to the clients new location if they connect to a different server. Walkabout proxies are similar, but don't require the client to explicitly say when it is disconnecting, as they do in JEDI. Elvin proxies [16] don't require explicit notification of disconnection, but they also don't support message migration between proxies.

Walkabout could potentially be implemented as a JMS [15] provider, as it provides a set of functions similar to those required by the API. Because it does not allow multiple clients to associate themselves with a given key, though, the topic-based pub/sub model of JMS is not applicable. The point-to-point model is more suitable, but queues must be created by an administrator. Under Walkabout, a client simply needs to register with any proxy once to create their 'queue'. Therefore, even though the interfaces are similar, their functionality is not close enough that Walkabout could be implemented as a JMS provider.

Some JMS implementations [9, 19] decentralise the messaging operations by placing gateway nodes at multiple points around a potentially wide-area network. However, none of them explore the benefits that could be experienced by placing these gateways in the same local networks as the individual communicating devices. This also appears to be true, to the best of our knowledge, of the other messaging systems discussed in this report.

Finally, while the peer-to-peer protocol of BitTorrent is suitable for transfers between large numbers of fixed systems, it is not optimised for transfers between a small number of highly mobile devices. Walkabout adapts the delivery rules of BitTorrent to make them more applicable to the sec-

ond situation, while still enabling peers to take advantage of parallel downloads wherever possible to increase transfer speeds.

## 7 Future work

The most important work that needs to be completed is the simulation, implementation and evaluation of the architecture. However, at some point in the future, the ideas presented in this section could be incorporated to improve the utility of Walkabout.

A layer of indirection could be introduced on top of client keys, so that a key is associated with a user and all the devices that they own, rather than just an individual device. A user registers the device (or devices) that they wish to use with the service at any given time, so that when someone sends them a message, this is where it will be delivered to. This idea could be extended further, to allow the user to specify that data directed to them be delivered to different devices, depending upon its properties. For example, they may wish to direct small files to their PDA, but large files to their laptop.

In the current system, message pieces may be deleted before they are ever delivered, due to cache constraints. Once a client believes it has completely uploaded a message, there is no way to recover from this without resending the entire thing. Therefore, it may be worthwhile to introduce network support for a receiving client or proxy to request individual pieces that are not available anywhere else directly from the original producer.

Prefetching has been shown to improve the performance of publish/subscribe with multiple event broker servers, by migrating subscriptions for a disconnected host to the server where it is expected to reconnect [2]. By tracking the movement patterns of a client, the Walkabout network could predict which proxy a disconnected client will next connect to, and prompt it to begin downloading the client's stored messages. If the client does indeed connect there, it can begin downloading immediately and will experience a faster transfer than in the non-predictive system.

Clients have key pairs, of which the public key is currently only used for identification. Signing and encryption with the private key would improve the security of the system. In particular, signing the header would enable receivers to check that the message they receive has not been tampered with.

Proxies can have a variety of parameters, and we would like to identify how tweaking each of them affects the overall transfer speeds.

Finally, Walkabout is vulnerable to malicious reconnections, where a client connects momentarily to a series of proxies, staying long enough to trigger message migration, but not to ever actually download any pieces. If a large num-

ber of clients did this, it could result in wasted network resources and an overall degradation in system performance. One possible way to restrict this sort of behaviour could be to enforce a cap on the number of connections per day.

## 8 Conclusion

In this report, we presented the design of Walkabout, a messaging architecture that improves the data transfer abilities of mobile devices across the Internet. It aims to maximise the throughput while a client is connected, which it achieves through the provision of an overlay network of proxies which: buffer uploads from producers; find the destination client(s); download messages on behalf of a consumer, even if they're not connected, so that they may download it when they reconnect; and migrate messages to follow consumers.

This overlay makes the system robust, as it will continue delivering a message no matter how much either the sender or the receiver moves, and will attempt to keep a message for a disconnected device for as long as possible.

The overlay also provides efficiency, because the destination proxy downloads pieces directly from the proxy that has them. The peer-to-peer download model makes use of multiple data sources wherever possible, to maximise the download speed at the destination proxy. This can lead to transfers that are faster than would be possible using other transfer paradigms, particularly as mobility increases.

We intimated the effectiveness of Walkabout through several scenarios, and we are currently in the process of evaluating them conclusively through simulation. After that, we intend to obtain real-life measures of performance, by producing and testing the applications that Walkabout enables.

## Acknowledgments

The authors would like to thank the Smart Internet Technology CRC for the support they provided to this research, and Mark Assad and Daniel Cutting for their valuable input.

## References

- [1] M. Assad, J. Kay, and B. Kummerfeld. The keep-in-touch system. In *Proceedings of Ubicomp 2005 Workshop on Situating Ubiquitous Computing in Everyday Life: Bridging the Social and Technical Divide*, Tokyo, Japan, September 2005.
- [2] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected operation in publish/subscribe middleware. In *Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04)*, Berkeley, California, USA, January 2004.
- [3] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. Technical Report CU-CS-944-03, Department of Computer Science, University of Colorado, January 2003.
- [4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.
- [5] B. Cohen. Incentives build robustness in BitTorrent. <http://www.bittorrent.com/bittorrentecon.pdf>, 2003.
- [6] B. Cohen. Bittorrent goes trackerless: Publishing with BitTorrent gets easier! <http://www.bittorrent.com/trackerless.html>, 2005.
- [7] G. Cugola, E. Di Notto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.
- [8] D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, January 1985.
- [9] S. Maffeis. An introduction to wireless JMS. White paper, <http://www.softwired-inc.com>, 2001.
- [10] P. Maymounkov and D. Mazieres. Kademia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS'02*, Cambridge, USA, March 2002.
- [11] C. E. Perkins. RFC 3344 - IP mobility support for IPv4, August 2002.
- [12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings ACM SIGCOMM'01*, San Diego, California, USA, August 2001.
- [13] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.
- [14] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Australia, September 1997.
- [15] Sun Microsystems. Java message service (JMS) API specification. <http://java.sun.com/products/jms/>.
- [16] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CC-Grid'01)*, Brisbane, Australia, May 2001.
- [17] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136 - dynamic updates in the domain name system (DNS update), April 1997.
- [18] E. Yoneki and J. Bacon. Gateway: A message hub with store-and-forward messaging in mobile networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, Providence, Rhode Island, USA, May 2003.
- [19] E. Yoneki and J. Bacon. Pronto: MobileGateway with publish-subscribe paradigm over wireless network. Technical Report UCAM-CL-TR-559, University of Cambridge Computer Laboratory, February 2003.

- [20] B. Y. Zhao, J. D. Kubiawicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, EECS, UC Berkeley, April 2001.

School of Information Technologies  
Madsen Building F09  
University of Sydney NSW 2006 AUSTRALIA  
T: +61 2 9351 4917 F: +61 2 9351 3838  
W: [www.alumni.it.usyd.edu.au](http://www.alumni.it.usyd.edu.au)

ISBN 1 86487 800 2