Enabling Cross-Technology Mobile Applications with Network-Aware References

Kevin Pinte, Dries Harnie*, and Theo D'Hondt

Software Languages Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
{kpinte,dharnie,tjdhondt}@vub.ac.be

Abstract. Mobile devices, such as smart phones, have become ubiquitous. This evolution has given rise to a vast ecosystem of mobile applications. Typically these applications only use a small subset of the networking technologies at their disposal. Building applications that use multiple networking technologies simultaneously or exploit knowledge about the available connections is a laborious task. Programmers must manually keep track of the connectivity state and duplicate communication code per connection type. This paper presents network-aware references, a distributed object-oriented programming abstraction that eases multi-networking for mobile applications and allows programmers to react to changes in the connectivity of different networks around them. We show how network-aware references are implemented and evaluate how well they switch between technologies.

Keywords: Network-awareness, mobile applications, multi-networking, distributed programming, Bluetooth, Wi-Fi

1 Introduction

In the recent years we have seen a boom in the market of mobile devices such as smart phones and tablets: they have become powerful enough to complement existing computers as internet devices. This evolution has proliferated all kinds of mobile applications for various tasks people need to do on the move. *Pervasive social applications* [1, 2] are a good example of mobile applications: they allow people to interact with their social network no matter where, even at social events themselves. Another example are peer-to-peer ad hoc multiplayer games as found on Nintendo's handheld DS console.

Next to the performance increase, mobile devices have also gained the ability to communicate using several different networking technologies: for example, an iPhone can communicate with other devices using short-range Bluetooth, medium-range Wi-Fi and long-range 3G technologies. More recently, devices such as the Samsung Nexus S have been released with the extremely short-range Near Field Communication (NFC) technology built-in.

Mobile applications typically exploit contextual information (e.g. location, proximity to other users, ...) to better anticipate the needs of users. *Network awareness* is an

* Funded by the Prospective Research For Brussels program of IWOIB-IRSIB.

integral part of context-awareness [3, 4]. Mobile applications can benefit from being network-aware since mobile devices are constantly on the move and their networking hardware allows them to "sense" devices and access points in the environment. Additionally, mobile applications often need to adapt the content they present to the user to the characteristics of the network connection that is currently being used [5].

We illustrate why network awareness is relevant for mobile applications by introducing a representative pervasive social application called Pixee that allows users to share and follow each other's picture libraries. The Pixee application gathers library information from nearby devices using Bluetooth and offers to follow other users if a library matches the user's profile.

The scenario goes as follows: on the way home from work, Alice and Bob, both Pixee users, happen to be in the same metro car and thus in Bluetooth range. Since they both like pictures of cats, Alice's picture library matches Bob's interests. When Bob arrives home, he opens the Pixee application and it presents him with a selection of picture libraries, including Alice's library. Bob accepts Pixee's suggestion and starts following Alice's picture library. Whenever Alice takes a new picture it is automatically uploaded to Bob. As Alice isn't home yet, Pixee uses her 3G connection to share pictures. In order to optimize for the networking technology used, Pixee resizes pictures before sharing them over 3G. When Bob rides on the metro to work the next day the Pixee application alerts him that Alice is nearby so they can meet in person and chat later. As long as their phones are in Bluetooth range, their phones will exchange high-resolution pictures.

Currently, various high-level middleware solutions exist that enable network awareness [5, 6]. Such solutions use network information to enforce quality of service (QoS): they adapt the application's network usage to optimize for the available bandwidth. However, such solutions do not support using multiple network links simultaneously; instead they choose one primary network link and use the other network links only as backup links. In the networking domain there is low-level support for multi-networking [7, 8]: they allow applications to connect to other devices using different networking technologies simultaneously, and switch seamlessly between these technologies as needed. Applications that use these multi-networking technologies are fully communication-agnostic: all communication is performed in an opaque way. As such, they do not allow the programmer to react to changes in the individual network links they encapsulate, making network awareness hard. Our goal is to provide mobile application programmers with a hybrid solution that allows programs to be network-aware while supporting multiple networking technologies simultaneously.

In this paper we focus on distributed object-oriented programming abstractions to develop mobile applications that exploit network awareness, like Pixee. Distributed object-oriented programming languages rely on the notion of *remote object references* to communicate with objects residing on other devices. Currently, these references use different, usually incompatible APIs for each networking technology. This leads to a lot of duplication: multiple references to the same object can exist simultaneously, potentially using different technologies. Programmers have to do manual bookkeeping to keep track of these different references.

We propose a new programming abstraction, *network-aware references*, that extends the concept of remote object references to abstract over the different networking

technologies used. In addition, we provide a high-level API that enables programmers to react to changes in network links and also allows them to control the different network technologies their applications use. The main contributions of this paper are:

1. We identify the challenges involved in developing network-aware programs (section 2);
2. We introduce network-aware references that abstract over the networking technology used (section 3);
3. We propose programmable *network behaviors* as a means for the programmer to adapt communication to the changing network situation (section 4);
4. We show how network-aware references are implemented (section 5) and evaluate their multi-networking aspects (section 6).

To conclude, we discuss existing approaches to network awareness and multi-networking technology in section 7.

2 Challenges in programming network-aware applications

As mentioned in the introduction, current mobile devices support several networking technologies. They exhibit different characteristics such as bandwidth, energy consumption, communication range, etc. These differences occur not only between technologies, but they can also be found between networking technologies of the same kind. For example, the free Wi-Fi network at an airport is restricted in bandwidth and usage volume compared to a wireless network at home.

To facilitate the development of network-aware mobile applications, several challenges need to be overcome:

C1. Abstraction over networking technologies A programmer should not be concerned with the low-level details of the networking technologies available. For example, setting up a connection using Bluetooth is very different from a Wi-Fi connection. Instead a unified interface to the different networking technologies should be provided. Communication with this unified interface should take advantage of all available network connections.

C2. Reactions upon network (un)availability The programmer should be able to detect the appearance and disappearance of network connections and react upon these events. Pixee's friend list reflects the connectivity status of the users: when Bob and Alice are connected via Bluetooth, Alice is notified that Bob is nearby so she can invite him for a chat. Additionally, the programs must be resilient to disconnections. The Pixee application seamlessly switches from Bluetooth to 3G when Alice leaves the metro.

C3. Dynamic adaptation of network behavior Although the low-level details of networking should be hidden from the programmer, he should retain high-level control over the networking technology used. For example, Bob's Pixee application resizes pictures when Alice's device is connected via a 3G link to limit the data transferred. Mobile applications can then take advantage of the unique properties of each networking technology.

In the next section we introduce network-aware references: a programming abstraction that tackles the above challenges.

3 Network-aware references (NARs)

Before we introduce network-aware references, we describe terminology used in distributed object-oriented languages. These languages use *proxy objects* to locally represent objects residing on remote devices. Proxy objects implement the same interface as the remote objects they represent, but they translate all method calls into remote method calls. The combination of a proxy object and its network link is called a *remote (object) reference*.

A program can acquire new remote references in three ways: 1) The remote object can be discovered using a peer-to-peer service discovery mechanism; 2) When peer-to-peer service discovery is not possible, clients can receive remote references from a globally reachable registry; 3) A remote reference is created when a local object is passed as an argument in a remote method call.

Network-aware references (NARs) abstract over the implementation details of different networking technologies and present a single reference to the programmer. A NAR consists of multiple remote references to the same object, each using a different network link. Every remote object is identified by a globally unique ID (GUID) based on the device or VM it is hosted on and an object identifier within that device or VM. When the application first discovers a new remote object, a NAR with a single reference is created. As additional references to the object are acquired, they are added to the NAR. Figure 1 compares traditional remote references and NARs graphically.

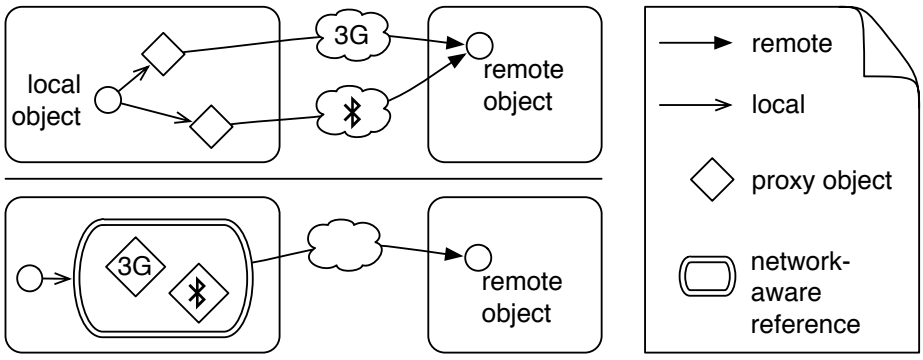

Fig. 1. A NAR encapsulates remote references to the same object.

In a mobile setting, network links are very unstable. They disconnect and reconnect as people move in and out of wireless communication range of others [9]. With traditional remote references (e.g. RMI [10]) remote method calls block and wait for a response. Furthermore, network disconnections are signaled as exceptions. Thus, this model is not suitable for mobile applications. We adopt the *far reference* model [11] instead, for two reasons. First, far references allow only asynchronous communication. A remote method call over a far reference immediately returns and the result, if any, can be retrieved using an asynchronous callback. Second, a far reference tracks the status of

the network link and can be in one of two states. It is either connected, in which it translates method calls in remote method calls. A far reference can also be disconnected, which means that it buffers all messages that are sent through it in a so-called "mailbox". Immediately after reconnection a far reference will try to transmit all outstanding messages in the mailbox to ensure no messages are lost.

Likewise, a NAR is disconnected when *all* of the underlying remote references are disconnected. As long as a NAR is disconnected, all messages sent to it are buffered in a unified mailbox. If one of the underlying remote references is reconnected or a new reference is added to the NAR, it switches back to the connected state. Figure 2 illustrates this: the NAR on the left is connected, as it encapsulates one connected remote reference using 3G and a disconnected remote reference using Bluetooth. The NAR on the right is disconnected, as all encapsulated remote references are disconnected.

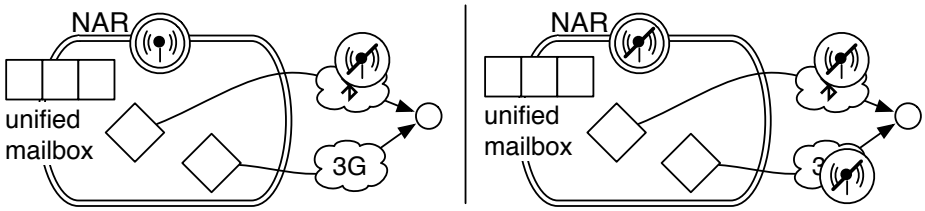

Fig. 2. Connection status of a NAR.

Whenever the programmer sends a message to a NAR, the system dispatches it to a remote reference from the set of references encapsulated by the NAR. The default behavior of a NAR nondeterministically selects a connected reference to transmit messages. Programmers can specify other behaviors by attaching a *network behavior* either to the NAR, or to individual messages. This network behavior then becomes responsible for transmitting messages using any of the references from the NAR. Currently, we offer a basic set of network behaviors that can for example limit message transmission to a certain technology or prioritize one technology over another. Network behaviors are further explained in subsection 4.2.

3.1 Communication Semantics

In this section we detail two properties of the communication semantics of NARs: they guarantee that messages sent to an object are processed in the order they are sent, and that messages are processed only once.

A NAR ensures the first property by serializing message sends through its unified inbox. Messages are processed one by one, and a message is only processed if the receiver acknowledges the receipt of the previous one in the queue. Figure 3 shows what happens if a message is sent to a NAR: first, it is added to the unified mailbox (1). The NAR constantly tries to deliver the first message in its mailbox. When the message eventually reaches the first position in the mailbox, it is removed from the queue

and marked with a serial number unique to the NAR (2), then the network behavior attached to the message selects a reference from the set of currently connected references (3). The message is then transmitted to the receiver using that reference (4). If an error occurs during the transmission or the network behavior does not choose a connected reference the message is returned to the front of the unified mailbox and the transmission is retried later (5).

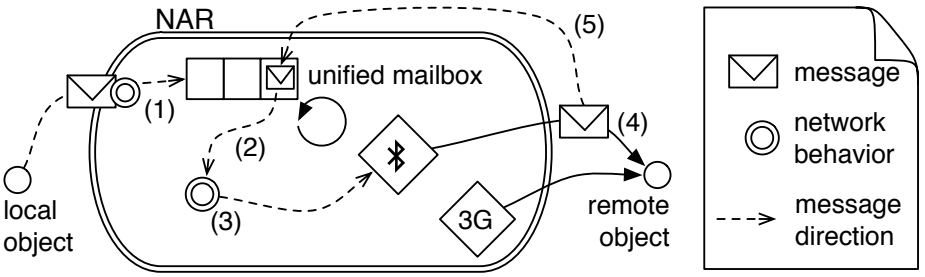

Fig. 3. The network behavior selects a remote object reference.

The receiving remote object processes all messages in the right order based on the serial number of the messages. In the case a message gets lost, messages with a higher serial number are not processed until the missing message arrives.

The second guarantee that NARs offer is that a message is processed only once. The default network behavior avoids message duplication by always selecting a single reference to transmit a message. However, programmers can build network behaviors that send duplicate messages intentionally, for example to ensure a critical message arrives as soon as possible. When a duplicate message arrives, the receiver first checks if it has already received a message with that serial number and ignores all duplicate messages.

4 NARs from a Programmers' Perspective

We have prototyped NARs in the distributed object-oriented programming language AmbientTalk [11]. AmbientTalk is designed to work in mobile settings and has already been used to build mobile applications such as a pervasive social application [12].

First, we show how to obtain a NAR. The following piece of AmbientTalk code uses the `whenever:discovered:`[1] language construct to install a handler that is called whenever a Pixee user is discovered in the environment. The block closure that is executed receives a NAR to the remote Pixee application as an argument in `aUser`. The handler requests the remote Pixee user's name, and adds the user to the buddy list. The left-arrow operator (←) represents an explicit asynchronous message send, while the dot operator is used for synchronous method invocation on local objects:

[1] NARs follow the AmbientTalk conventions: `when:` constructs are deactivated after triggering once, whereas `whenever:` constructs trigger every time.

```
whenever: PixeeUser discovered: { |aUser|
  when: aUser←getName() becomes: { |name|
    GUI.addUser(aUser, name) } }
```

The asynchronous call to `getName()` immediately results in a *future*: a placeholder for a future value. The `when:becomes:` construct then installs a handler that is executed when the future is resolved with a return value.

In the remainder of this section we will show how programmers can respond to changes in the network availability around them, adapt the network communication to certain networking technologies, and implement their own network behaviors. We will demonstrate how the Pixee application uses the API offered by NARs. The examples we present here use AmbientTalk syntax, but NARs can be implemented in other distributed object-oriented systems, like RMI [10].

4.1 Network Availability

As mentioned in the introduction, information about the available networks forms an important source of context for mobile applications. A NAR can encapsulate several remote references, so we offer a `linksOf:` primitive that returns a snapshot of the state of these references to the programmer.

For example, Pixee allows users to explicitly share a picture with another user who is close by, to highlight certain photos in their library. The user interface only allows this if the other user is connected via Bluetooth:

```
if: (linksOf: aUser).contains(Bluetooth) then: {
  GUI.showShareButtonFor(aUser) }
```

Pixee also shows that *changes* in the network connectivity of references play a big role, as users move about and connectivity fluctuates. For example, Pixee's friend list reflects the connectivity of a user's friends and updates it in real time. It uses the `linksOf:` primitive above to draw the friend list initially and then updates the *nearby* status of individual friends whenever their connectivity changes:

```
whenever: aFriend linkStatusChanged: { |change|
  if: (change.link == Bluetooth) then: {
    if: (change.isConnected) then: {
      GUI.notifyFriendNearby(aFriend);
    } else: {
      GUI.notifyFriendLeaving(aFriend) } } }
```

The `whenever:linkStatusChanged:` function installs a handler that is called whenever one of the references in a NAR disconnects or reconnects. It receives a *change* object as an argument. This object contains the link of which the connection status changed and the new status of the link.

We also provide two generalized handler functions that are triggered when a NAR switches to a disconnected or a connected state, respectively. Programmers can react on disconnections using a `whenever:disconnected:` handler. Likewise, programmers can use `whenever:reconnected:` to react on reconnections, which happen whenever a new reference is added to a disconnected NAR or one of the existing references it encapsulates is reconnected. The example below shows adding or removing a user from the friend list in the user interface when this user reconnects or disconnects:

```
whenever: aFriend disconnected: { GUI.hide(aFriend) };
whenever: aFriend reconnected: { GUI.show(aFriend) };
```

Together with the `linksOf:` primitive, these event handlers provide programmers with information about the connectivity of the different references in their applications and allow them to react on changes in network connectivity.

4.2 Network Behavior Adaptation

In order to allow programmers to steer communication towards certain networking technologies, we introduce *network behaviors*. Our implementation provides a number of built-in network behaviors. Additionally, programmers can implement their own behaviors and override the default behavior attached to NARs using *annotations*.

The first network behavior is called **Only**: it restricts message transmission to a set of network technologies. The example below ensures an explicitly shared picture is only sent using Bluetooth:

```
aUser←sendPicture(aPicture)@Only(Bluetooth)
```

This message will *only* be sent using a Bluetooth connection: if there is no connected Bluetooth reference to `aUser` the message is returned to the mailbox and message processing for the **aUser** NAR will stop until a Bluetooth reference is connected.

The **Prefer** behavior allows programmers to order an arbitrary number of technologies in decreasing order of preference:

```
aFriend←sendPicture(aPicture)@Prefer(Bluetooth, WiFi)
```

The **Prefer** behavior here first tries to transmit the **sendPicture** message using a Bluetooth link; if this is not available it tries to use a Wi-Fi link. If a Wi-Fi link is also unavailable, the **Prefer** behavior defers to the default behavior: this will either transmit the message using other technologies, or buffer the message if the NAR is disconnected.

Programmers can override the default network behavior for the whole application or just for a specific NAR using the following primitive functions:

```
setDefaultBehavior: aBehavior;                // application-wide
setDefaultBehavior: aBehavior for: aReference; // for the given reference
```

This operation makes every message sent using the NAR use the `aBehavior` behavior, unless programmers explicitly override this behavior by annotating messages with another behavior.

Existing network-aware approaches have shown that working with properties instead of explicit technologies is more versatile. For example, if a programmer wants to send a big file using a "fast" network connection, he should be able to use a keyword like **Fast** instead of listing every "fast" network connection explicitly. In order to do this, we provide programmers with a set of *categorization functions* that select networking technologies based on their properties.

We provide three built-in categorization functions: we offer a categorization function **Speed(x)** which only selects network links that theoretically offer at least **x** Mbits of bandwidth. The second categorization function is **Secure**, which selects network links

that only do point-to-point communication or encrypt sent messages. Finally, cost can also be a factor in deciding which network link to use: we provide a **Free** categorization function that only selects network links that do not cost money to use. Currently these properties are statically defined as attributes of the network links.

Programmers can create their own categorization functions by refining the built-in ones described above. For example, they can define **Fast** as **Speed(10)**. Additionally, programmers can manually define new categorization functions based on the properties of the network links.

Using categorization functions instead of explicit technologies has another advantage: when a new networking technology becomes available, one only has to declare to which categories that new technology belongs or create new categorization functions if necessary. For example, a body area network (a very close-range wireless technology for non-intrusive health monitoring [13]) could be deemed **Fast** and **Secure**, but could also introduce a new category such as **PhysicalContact**.

4.3 Writing custom network behaviors

The network behaviors we have presented so far only influence the technology selection process. If we want to allow full network awareness, programmers also need to be able to adapt the content of messages to the technology used to transmit them. In our system, behaviors are represented by objects that expose a single method `transmit(connections, message)`. This method is invoked during step three of the message sending process (when network technologies are chosen). This method transmits the message(s) to the receiver and informs the NAR that the message can be removed from the mailbox. The `connections` and `message` parameters of the method are the current set of available network links in the NAR and the message being sent, respectively.

In the scenario, Pixee resizes pictures before transmission if a Bluetooth link was not available; programmers can implement this as follows:

```
1  def PictureResizer := extend: Prefer(Bluetooth, WiFi) with: {
2    def transmit(connections, message) {
3      def btLink := connections.find: { |x| x.link == Bluetooth };
4      if: (btLink != nil) then: {
5          message.arguments := message.arguments.map: { |arg|
6            if: (isPicture(arg)) then: { arg.resize() } else: { arg } };
7      super^transmit(connections, message) } };
```

This behavior extends the **Prefer** behavior and first looks for Bluetooth links in the network links managed by the `aFriend` NAR. If there are no Bluetooth links available, the behavior replaces pictures in the arguments list of the message with a resized version (lines 5–6). Finally, the message (line 7) is transmitted by the inherited **Prefer** behavior.

We also provide a `behavior:` construct to create a new behavior that inherits from `defaultBehavior`. All transmit calls are delegated to the default behavior eventually, since it enforces the two properties we outlined in section 3 (message ordering and no processing of duplicated messages).

5 Implementation

In this section we will discuss the implementation of network-aware references. We will first explain how the networking subsystem of AmbientTalk is structured and how this is exposed to programmers in subsection 5.1. We will then show how network-aware references are implemented on top of this in subsection 5.2

5.1 The AmbientTalk networking subsystem

Originally, the AmbientTalk VM was tightly coupled to its networking implementation, limiting it to only one network interface using the TCP/IP protocol. This network interface was represented by a single *communication bus*, which performs three duties for AmbientTalk: a) discovering objects on devices that export them using the same technology; b) marshalling communication and ensuring message ordering; c) signaling disconnections and reconnections.

We have decoupled the networking subsystem from the rest of the system, to allow other networking technologies to be plugged in easily. AmbientTalk now maintains a set of communication buses, one per network address (a network interface can respond to multiple addresses). For example, a typical smart phone will have a Bluetooth communication bus and a TCP/IP communication bus for the Wi-Fi interface.

The three duties of a communication bus influence the status of connected references. For example, the TCP/IP communication bus uses *heartbeat* packets sent via multicast UDP to determine the connectivity of other hosts. New devices are discovered as soon as they send their first heartbeat, and disconnections are signaled when either the heartbeat has been absent for a given amount of time or a communication error occurs during message transmission. This is different from the Bluetooth communication bus, where device discovery can take up to 12 seconds and is thus only done intermittently. For the Bluetooth bus the leading cause of disconnections will be communication errors signaled during message transmission.

Each communication bus is also associated with a *network link* object, which programmers can use to interact with that specific network interface. Each network link can be used to enable or disable its communication bus and set up discovery handlers. For example, discovering objects exclusively on the Bluetooth network link:

```
whenever: Service on: Bluetooth discovered: { |ref|
  system.println("Discovered: " + ref);
  whenever: ref disconnected: {
    system.println("Disconnected: " + ref); } }
```

Here `Bluetooth` is bound to the Bluetooth network link object. This snippet sets up a discovery handler that is invoked whenever an object with the nominal type `Service` is discovered using the Bluetooth communication bus. This discovery handler is then called with a *far reference* as parameter, which is a local proxy for the remote object. The far reference is associated with the network link that created it, the object it references and the VM this object resides on.
The programmer can also publish an object on a specific network link:

```
export: anObject as: Service on: Bluetooth;
```

In the interest of backwards compatibility, the discovery and publish constructs found in AmbientTalk (`when:discovered:`, `whenever:discovered:` and `export:as:`) invoke their network-aware counterparts on all available network links. The disconnection and reconnection handlers (like above) operate on pre-existing references so programmers do not need to specify network links.

At this stage there is only one technology per far reference, so messages sent to a far reference can only travel along a single path. This entails that discovering the same service using two different network technologies will result in two far references, as described in section 3.

5.2 The architecture of network-aware references

With these modifications to the networking subsystem, AmbientTalk can communicate using different technologies but suffers from the challenges we identified in section 2. Our implementation of network-aware references tackles these challenges as follows:

C1. Abstraction over networking technologies The previous subsection already outlined how the networking subsystem in AmbientTalk can abstract over the communication technology used. However, references are still created per technology. When programmers import the NAR module, it replaces the built-in discovery operations as follows. When the programmer issues a `whenever:discovered:` statement it is translated into several `whenever:on:discovered:` statements, one per network link. When one of the discovery statements are triggered, they first check if a NAR for the discovered object already exists. If so, the reference is added to the NAR and the user-supplied discovery block is not triggered. Any `whenever:linkStatusChanged:` handlers registered on the NAR are triggered.
If no NAR exists, a new NAR is created which contains just that reference. Finally, the NAR is added to a table with the GUID of the discovered object as key.

C2. Reactions upon network availability As part of the discovery process, the NARs module also installs disconnection and reconnection handlers. Whenever the networking subsystem detects that a reference has disconnected or subsequently reconnected, it signals this change to the NAR. The NAR in turn signals the `whenever:linkStatusChanged:` message. If the last reference in a NAR becomes disconnected, the NAR as a whole becomes disconnected and triggers all installed disconnection handlers. Vice versa, if one of the references in a disconnected NAR reconnects, the NAR is reconnected and all installed reconnection handlers are triggered. After reconnection, a NAR retries transmission of messages in the queue.

C3. Dynamic adaptation of network behavior As we explained earlier, NARs allow programmers to specify the network behavior of their communication. Every message submitted to a NAR is put into a message queue, which are processed one by one. If the network behavior of the first message in the queue is not amenable to transmission the queue is blocked and no messages are transmitted (e.g. if a

message has a **Only** annotation and the desired network link is not online). Transmission of the message queue is retried whenever a new reference is added to a NAR or an already-added reference comes back online.

6 Evaluation

In this section we demonstrate the behavior of network-aware references in the face of partial disconnections. As mentioned in the introduction, mobile devices nowadays can communicate using more than one wireless communication technology and the system should always pick the "best" interface. If this interface disconnects due to a communication error or the other party moving out of range, the system should switch to a different technology. No communication should be lost during this switch and the handover time (the time where no data is sent) should be kept to a minimum.

The scenario we test is inspired by the "mobile connectivity" scenario in [14]: two smart phones that discover each other in the environment. One phone runs a *receiver* service, the other phone runs a *sender*. At time step `t0` the sender starts sending messages to the receiver at a pace of 10 messages per second. At time step `t1` the Wi-Fi connectivity is temporarily interrupted and at time step `t2` Wi-Fi connectivity is restored. This timeline is illustrated at the top of Figure 4.

First, we reconstruct the original scenario from [14] where the sender only sends messages to the receiver using Wi-Fi technology. We use the **Only** network behavior so that the `ping` message is only transmitted over a Wi-Fi link:

```
when: MobileConnectivityReceiver discovered: { |receiver|
  whenever: millisec(100) elapsed: {
    receiver←ping()@Only(WiFi); } }
```

When the Wi-Fi link becomes disconnected at `t1` all messages being sent are buffered at the sender. At `t2` connectivity is resumed, and the accumulated messages are flushed to the receiver. This behavior is illustrated in the middle graph of Figure 4. The spike in the timeline only happens 1.5 seconds (on average) after `t2` because the devices wait for heartbeats, as explained in the previous section. For Bluetooth links this reconnection process takes upwards of 15 seconds (on average), depending on the number of devices in communication range.

Our second implementation demonstrates the multi-networking facilities of NARs and will make use of both Wi-Fi and Bluetooth links to send messages. We define the "best" interface by annotating the `ping` message with the **Prefer** network behavior:

```
when: MobileConnectivityReceiver discovered: { |receiver|
  whenever: millisec(100) elapsed: {
    receiver←ping()@Prefer(WiFi, Bluetooth); } }
```

Before the Wi-Fi connectivity is interrupted, the behavior of the `ping` message will select the Wi-Fi link over the Bluetooth link to send the message. At time step `t1` the behavior can no longer select the Wi-Fi link and it selects the Bluetooth link instead. The bottom graph in Figure 4 shows how the Wi-Fi→Bluetooth handover occurs almost instantly. When the Wi-Fi link reconnects, the behavior will again prefer it over the Bluetooth link, explaining the handover at the 16–17 second mark. As before, the time

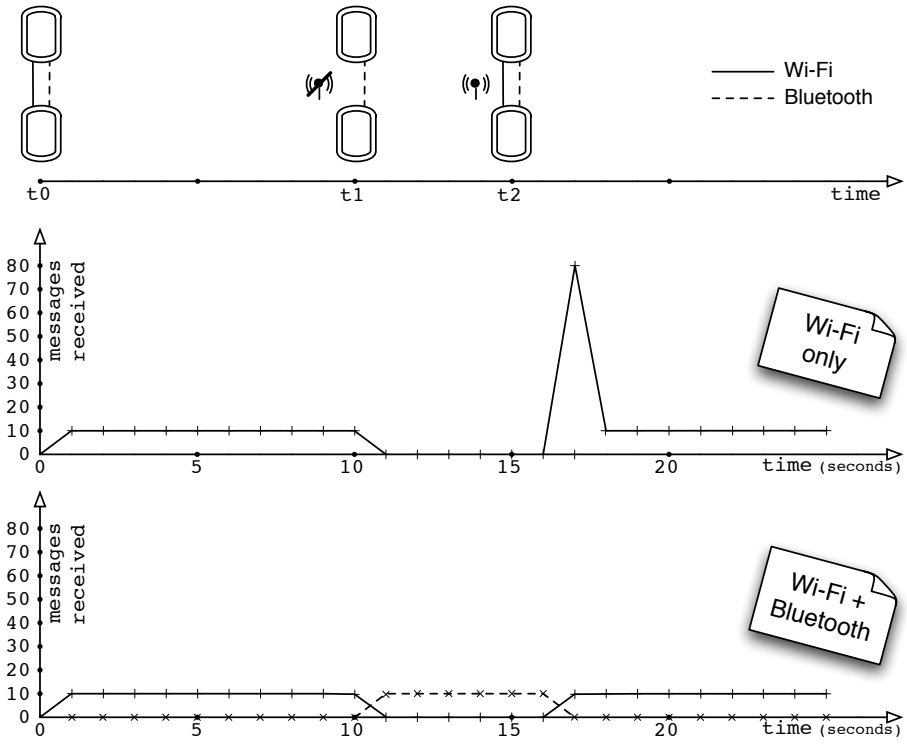

Fig. 4. Mobile connectivity scenario (top); benchmarks: only Wi-Fi (mid), Wi-Fi & BT (bottom)

gap between reestablishing Wi-Fi connectivity and the Bluetooth→Wi-Fi handover is due to the discovery and reconnection process. Note that message transmission is not interrupted at any point during this experiment.

7 Related work

In this section we discuss how network-aware references fit into the state of the art. Traditionally, network-aware applications are defined as "applications that adapt to network conditions in an application specific way" [6]. This has led to frameworks for maintaining quality of service (QoS) in media streams [5], where image or sound quality is reduced if the available bandwidth decreases. These are usually implemented in a framework or middleware and require the programmer to set up policies, giving up explicit control. Current network-aware applications assume network links are stable and treat network failures as an exceptional case. This makes them unsuitable for a mobile situation where pervasive social applications are deployed.

A number of network protocols have been proposed that enable multi-networking at a low level, like SCTP, mSCTP, SIGMA, TraSH and Mobile IP(v6) [7, 8]. However, these approaches focus on the problem of ensuring devices are always reachable at a

certain address and maintaining existing network connections when a mobile device migrates to a different access point (horizontal handover). Some technologies support transitions between different networking technologies (vertical handover), but they assume these transitions are short-lived, so they limit themselves to ensuring no data is lost during a transition. In contrast, NARs accept that there may be multiple connections at once, which can break at any time. NARs still ensure that all communication arrives, but cannot offer time guarantees: a message with the **Only** behavior attached will only be sent when a Bluetooth connection is available. This may depend on user mobility.

In [15] a policy-based architecture is proposed that manages several different routes to another device. They use policies to select appropriate network interfaces and set priorities between interfaces if several policies apply. However, their approach is not suitable for a mobile setting as their system immediately returns an error if there are no matching interfaces at the moment a packet is sent. In contrast, NARs buffer communication until a connection is re-established. Furthermore, their policies currently operate at the system level instead of the application level, so all applications on the system have to agree on the same set of policies.

Haggle [16] is an architecture that enables seamless network connectivity for devices in dynamic mobile environments. It abstracts over different network transport bindings and protocols so that applications become communication agnostic. Haggle is a central component in the mobile device that selects and switches network links as needed. In contrast to using NARs, programmers have no control over communication within a single application and cannot adapt its behavior to changes in the network context.

8 Conclusion and future work

In this paper we have introduced network-aware references (NARs): a programming abstraction that encapsulates several references to remote objects over different networking technologies and keeps track of the state of the network links involved. Network-aware references tackle the three challenges for programming network-aware applications we listed earlier: abstraction over networking technologies, reacting to changes in the network availability, and dynamically adapting network behavior. We have demonstrated how network-aware references can be used to build mobile applications using a representative pervasive social picture-sharing application called Pixee.

We are currently exploring different types of network behaviors and how they interact with NARs as they are defined here. For example, a behavior that limits retransmission of messages, or a behavior that spreads parts of a message across different links. Secondly, we would like to make the information offered to our system, like speed, communication range, pricing, etc. more dynamic. An application using Wi-Fi could then automatically switch to a different technology as the user enters an airport where wireless is not free to use. Additionally, we intend to allow these parameters to vary per reference rather than per network link (e.g. the signal strength for a reference over Bluetooth). Finally, in the search for related work we discovered heterogenous routing: transmitting packets in peer-to-peer networks where not all nodes speak the same protocol. We are currently investigating if NARs can be adapted for this.

References

[1] S. Ben Mokhtar and L. Capra, "From pervasive to social computing: Algorithms and deployments," in *ACM Inter. Conf. on Pervasive Services (ICPS)*, 2009.

[2] A. Meshhadi, S. Ben Mokhtar, and L. Capra, "Habit: Leveraging human mobility and social network for efficient content dissemination in manets," in *IEEE Inter. Symp. on a World of Wireless, Mobile and Multimedia Networks*, 2009.

[3] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *First Workshop on Mobile Computing Systems and Applications*, pp. 85–90, 1994.

[4] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *1st Intl. Symp. on Handheld and Ubiquitous Computing (HUC)*, (London, UK), pp. 304–307, Springer-Verlag, 1999.

[5] J. Bolliger and T. Gross, "A framework-based approach to the development of network-aware applications," *IEEE transactions on Software Engineering*, vol. 24, no. 5, 1998.

[6] N. Miller and P. Steenkiste, "Collecting network status information for network-aware applications," in *IEEE INFOCOM*, vol. 2, pp. 641–650, Citeseer, 2000.

[7] S. Fu, M. Atiquzzaman, L. Ma, and Y. Lee, "Signaling cost and performance of SIGMA: A seamless handover scheme for data networks," *Wireless Communications and Mobile Computing*, vol. 5, no. 7, pp. 825–845, 2005.

[8] W. Xing, H. Karl, A. Wolisz, and H. Müller, "M-SCTP: Design and prototypical implementation of an end-to-end mobility concept," in *Proc. 5th Intl. Workshop The Internet Challenge: Technology and Applications, Berlin, Germany*, Citeseer, 2002.

[9] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-Oriented Programming," in *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 2005.

[10] T. Downing, *Java RMI: remote method invocation*. IDG Books Worldwide, Inc. Foster City, CA, USA, 1998.

[11] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter, "Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks," in *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pp. 3–12, IEEE Computer Society, 2007.

[12] E. Gonzalez Boix, A. Lombide Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, and T. D'Hondt, "Flocks: Enabling dynamic group interactions in mobile social networking applications," in *SAC 2011: the 26th Symposium On Applied Computing – Mobile Computing and Applications track (to appear)*, 2011.

[13] E. Jovanov, A. Milenkovic, C. Otto, and P. De Groen, "A wireless body area network of intelligent motion sensors for computer assisted physical rehabilitation," *Journal of NeuroEngineering and Rehabilitation*, vol. 2, no. 1, p. 6, 2005.

[14] J. Collins and R. Bagrodia, "Programming in mobile ad hoc networks," in *4th Annual International Conference on Wireless Internet (WICON '08)*, pp. 1–9, 2008.

[15] J. Ylitalo, T. Jokikyyny, T. Kauppinen, A. Tuominen, and J. Laine, "Dynamic network interface selection in multihomed mobile hosts," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, 2003*, Published by the IEEE Computer Society, 2003.

[16] J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. H. Lim, and E. Upton, "Haggle: seamless networking for mobile applications," in *UbiComp '07: Proc. of the 9th international conference on Ubiquitous computing*, (Berlin, Heidelberg), pp. 391–408, Springer-Verlag, 2007.