



IST-002057 PalCom

Palpable Computing:

*A new perspective on
Ambient Computing*

Deliverable 41 (2.4.3) Components and Communication

Due date of deliverable: m 36
Actual submission date: m 36

Start date of project: 01.01.04
Duration: 4 years

Ecole Polytechnique Fédérale de Lausanne (EPFL)

Revision: 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

**Integrated Project
Information Society Technologies**



Contents

1	Introduction	4
1.1	Workpackage Objectives	4
1.2	Contributors	5
1.3	Audience	5
1.4	Comments	5
2	PalCom Communication Model	6
3	Addressing	7
3.1	PalCom URL Addresses	7
3.2	PalCom URN Addresses	7
4	Message Formats	8
4.1	Message nodes	8
4.2	Message composition	8
4.3	Message data nodes	9
4.4	Message header nodes	9
5	Communication Model Layers	10
5.1	Media Abstraction Layer	10
5.2	RASCAL Layer	10
5.3	Routing Layer	11
5.4	Communication Layer	12
5.4.1	Publish/Subscribe Communication Component	12
5.4.2	Point-to-Point Communication Component	13
5.5	Function Layer	17
5.5.1	Discovery	17
5.5.2	Data Streaming	22
5.6	Service Layer	22

6	Open Issues	22
6.1	Discovery	24
6.1.1	Architectural Elements to be Discoverable	24
6.1.2	Large Numbers of Discoverable Elements	24
6.1.3	Discovery in Complex Networks	24
6.2	Open issues for Routing	24
6.3	Quality of Service	26
7	Conclusions	27
7.1	Task 1: Validation of the communication and discovery model and implementation of other core components	27
7.2	Task 2: Further improvement of the communication component and improved codebase for the application prototypes	27
7.3	Task 3: Specification of the communication model & other core components	27
	References	28
A	Complete message format specification	29
B	Message Formats for PalCom Discovery Messages	30
B.1	Request messages	30
B.2	DeviceInfo	30
B.3	ServiceListRequest	31
B.4	ServiceList	32
B.5	ServiceDescriptionRequest	33
B.6	ServiceDescription	33
B.7	ConnectionListRequest	34
B.8	ConnectionList	34
B.9	RemoteConnect	35
B.10	RemoteDisconnect	35
C	Code example with connections	36

1 Introduction

This deliverable elaborates on the communication model for Palpable Computing, as previously described in [8, 9, 10].

The deliverable constitutes an update of Part II of Deliverable 23, “Specification of Component & Communication Model” [10]. It describes the general format of messages (Section 4) and the different types of messages that are used in the different components of the communication model. PalCom’s layered and modularized communication model is presented, including publish/subscribe and point-to-point communication components in the Communication layer (Section 5.4), and routing in the Routing layer (Section 5.3), and discovery in the Function layer (Section 5.5).

Appendix A contains the complete specification of the message format. Appendix B documents the formats of messages used in the PalCom discovery protocol. Finally, Appendix C gives a small example of code that uses connections, described in Section 5.4.2.

1.1 Workpackage Objectives

Objectives

This WP focuses on specification and development of the core components in the PalCom architecture. A primary objective is the refinement and stabilization of the current model for discovery of PalCom services and the subsequent communication between them. Due to the nature of palpable systems it will have to deal with the heterogeneous and highly dynamic underlying communication media. Specification and implementation of other core components; suggestions for these components are storage, simple GUI and logging. The components to be implemented will be identified in cooperation with WP7-12 and will be used in the application prototypes developed in those WPs.

The specification of the Core components, including the communication and discovery model, will be part of the contribution to the open architecture specification, developed in WP2. The reference implementations of the identified components are implemented on top of the palpable runtime environment developed in WP3. Finally, WP4 will provide support for the components designed in WP5 and WP6.

Potential research issues include:

- Routing and service discovery in heterogeneous networks
- Quality of service in ad-hoc networks
- Resource-aware communication primitives
- Programming language support for communication and discovery
- Distributed logging in a heterogeneous network

Tasks

Task 1: Validation of the communication and discovery model and implementation of other core components

The purpose of this task is to validate the current implemented communication and discovery model. The validation will be done in close cooperation with WP7-12, special attentions will be put into the use of the model in heterogeneous networks. Parallel with the validation implementation of core components identified in cooperation with WP7-12 will be started. The components should provide a solid foundation for the requirements of the application prototypes developed in WP7-12. The components will also be part of the open source toolbox. (IM33)

Task 2: Further improvement of the communication component and improved codebase for the application prototypes

The fulfilment of this task will provide a communication component and discovery mechanism able to act in heterogeneous networks and provide improvements to the core components started in the previous task. (IM35)

Task 3: Specification of the communication model & other core components

The purpose of this task is to create a specification of the communication and discovery model, and specifications of the other core components. Focus will be on mechanisms and techniques for handling communication and discovery in loosely coupled systems, and for support of several palpable qualities: composition/decomposition (e.g. connections), visibility/inspection (of devices, services, connections), stability, understandability and user-control of discovery and communication processes otherwise being automated through sense-making & negotiation when changes occur. This task also includes identification of mechanisms needed to support contingency- and resource handling in WP5 and end-user composition in WP6. The result of this task will be Deliverable 2.4.3, end of month 36. (D2.4.3)

Deliverables*Month 36: Specification of core components (2.4.3)*

This deliverable will provide detailed specification of the core components including the communication and discovery model. Focus will be on mechanisms and techniques for handling communication and discovery in loosely coupled systems. This deliverable includes identification of mechanisms needed to support contingency- and resource handling in WP5 and end-user composition in WP6.

Several PalCom challenges are potentially addressed in this deliverable: Composition and decomposition (e.g. of connections), visibility/inspection (of devices, services, connections), stability, understandability and user-control of discovery and communication processes otherwise being automated through sense-making & negotiation when changes occur.

1.2 Contributors

The following people have contributed to this deliverable:

- Boris Magnusson, Lund University,
- David Svensson, Lund University,
- Jacob Frølund, University of Aarhus,
- Nikolay Mihaylov, Ecole Polytechnique Fédérale de Lausanne (EPFL),
- Roberto Ghizzioli, Whitestein Technologies AG,

Contributors to previous revisions include:

- Giovanni Rimassa, Whitestein Technologies AG,
- Michael Lassen, University of Aarhus,
- Tony Gjerlufsen, University of Aarhus,
- Peter Ørbæk, University of Aarhus,
- Peter Andersen, University of Aarhus,

1.3 Audience

The target audience of this document are developers of palpable systems. It describes the structure, protocols and message formats of the PalCom communication model, and the programming interfaces of the components that implement them.

1.4 Comments

Please direct comments to [or](#) to the individual contributors.

2 PalCom Communication Model

This chapter describes the details of the communication model of the PalCom Open Architecture, defines the different parts that make up the communication model, the use of those parts, and discusses the detailed structure of some of these parts and their protocols.

The communication model is a fundamental aspect of the architecture. It allows the deployed and running services to collaborate with each other by firstly discovering each other, and thereafter allowing the services to communicate individually or in groups, locally on a device or between multiple physically separated devices.

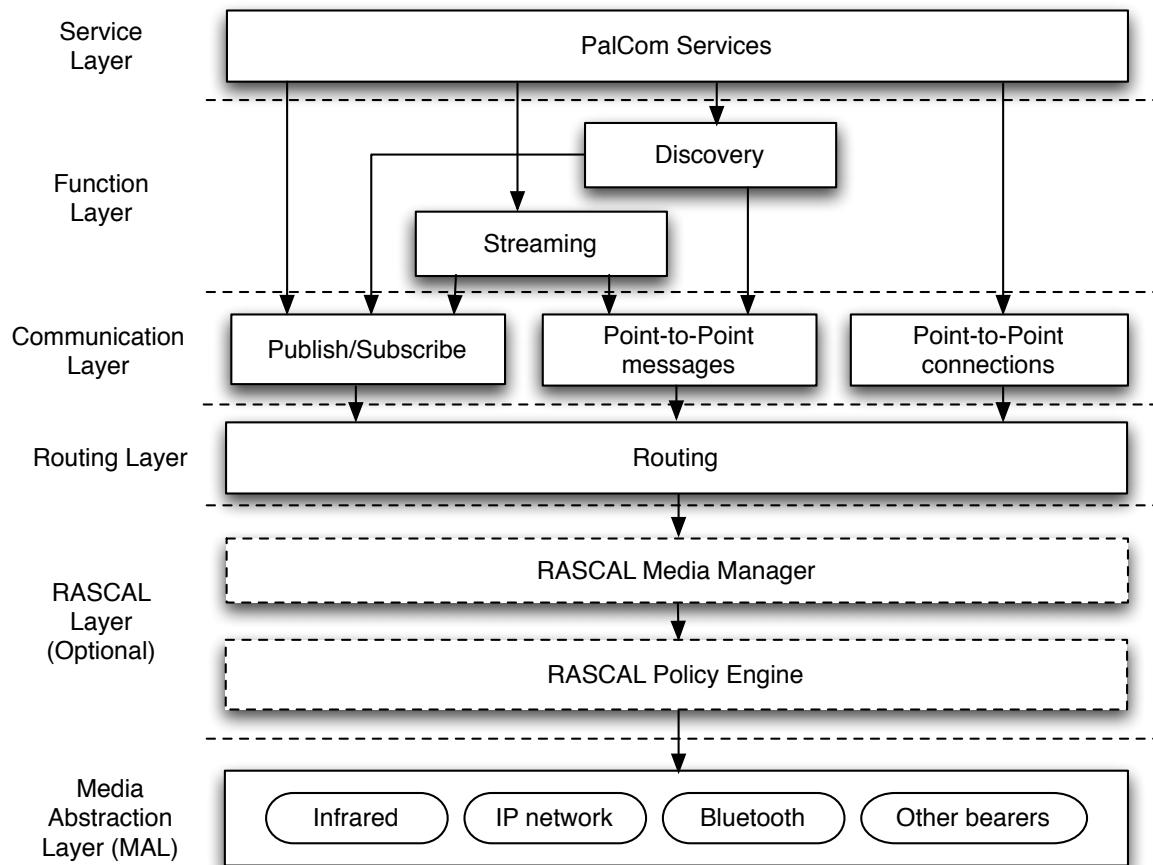


Figure 1: The layered, modularized communication model

The main goal of the communication model is to offer a simple, yet powerful model to the services providing:

- *Portability* – services written in different programming languages, running on different hardware platforms are able to communicate using a shared communication protocol on top of an available network technology.
- *Applicability* – an API that provides the mechanisms commonly needed by the service developer. Yet easy-to-use and well-designed, shielding the complexity of the underlying networking technologies.
- *Robustness* – partial failures of the distributed system are handled as they are detected, and the loss of system capability is proportional to the size of the faulty part. Moreover, suitable contingency mechanisms are in place to ensure that the system reconfigures itself to minimize the impact of faults.

- *Modularity* – a modular design allowing for scaling the communication capabilities going from simple communication capabilities on resource constrained devices to rich communication capabilities on advanced devices.

The communication model is split into a number of layers, each consisting of a number of communication components. Each layer provides a level of abstraction, which hides implementation details from the components at the higher-level layers. The layered communication model is illustrated in Figure 1.

The logical abstractions provided by the layers may or may not be observed by higher-level layers, including services. A service may decide to use a low-level layer rather than the layer directly below it, if it provides a better fit for its communication requirements. This, however, should be avoided since the clients of such service will have to understand its protocol.

Not all components in each layer are mandatory. Components may be present in the PalCom runtime environment according to its needs, resource and hardware capabilities. However, if a particular component is present, then all components on which it depends have to be present too.

3 Addressing

Addresses are used for specifying the counterparts in communication. We distinguish between location-specific and location-independent addresses. For the former we utilize Uniform Resource Locators (URL) [6]. For the latter - Uniform Resource Names (URN) [5, 7].

3.1 PalCom URL Addresses

The addresses follow the familiar URL style, which has the advantage of being printable and human-readable. A PalCom URL contains a protocol-specific part that is sufficient to reach the communication component at the remote side. Another part is used to specify the receiving service there. For that part, a service identifier is too coarse-grained: it should be possible for one service to receive messages on more than one address, instead of having to handle internal distribution of messages using data in the messages. Therefore, the notion of a *selector* has been introduced. This is a string, assigned dynamically to a receiver by the point-to-point communication component, which identifies the receiving service. The selector becomes the second part of the address, and is not the same as a UDP or TCP port, but used for any transport protocol. All selectors must be unique on a given device.

This is an example of a unicast address, with UDP as the transport protocol and 12 as the selector:

```
palcom: datagram://10.0.0.1:1024/12
          protocol-specific part
```

3.2 PalCom URN Addresses

When a device or a service is not located at a fixed address but becomes known using the mechanisms of discovery (Section 5.5.1) it can be accessed using location-independent URN.

```
urn:palcom://mydevice/service1
```

The Communication Layer (Section 5.4) is responsible for maintaining the mapping from URN to URL.

4 Message Formats

PalCom defines a general wire format for messages. This format is used for both control messages and data message exchange between services, except for those services that use their own, private message formats. Those special services implement their communication against the Media Abstraction Layer directly.

The message format has been selected so that it permits hierarchically structured messages, with messages inside messages at any number of levels. Another important point is that the messages should be human-readable, as far as possible. That makes inspection and debugging easier. Finally, care has been taken to keep messages compact, and make parsing simple.

Complete description of the message format can be found in Appendix A.

4.1 Message nodes

Messages are structured as sequences of message nodes. The general layout of all message nodes is given by the following EBNF grammar (where ? after a symbol means zero or one occurrence, and * means zero or more occurrences):

```

MessageNode ::= Descriptor Data.
Descriptor ::= Format ';' Length ';' '.
Format ::= Byte.
Length ::= Number.
Data ::= Byte*.

```

where:

- `' ; '` – field separator.
- `Format` – one byte format identifier; usually an ASCII letter or symbol.
- `Length` – the ASCII representation of the length of `Data` in bytes.
- `Data` – the content of the message node; its structure depends on the format identifier.

Calculating the length involves taking the length of `Data`, but not including the `Descriptor` parts in the length. This means that the reader of a message needs to parse the `Descriptor` parts and then add the length to the current position in order to get past the current message node, and thus to the start of the next message node.

Currently, the following message nodes have been defined: `SingleMessage`, `Multipart`, `Version`, `RoutingR`, `RoutingS`, `Mark`, `Connection`, `Chopped`, `Reliable`, `Connection`, `AckMessage`, `ResendMessage`.

4.2 Message composition

A message is composed of a header, with a number of message nodes, followed by the actual data in the last message node. All nodes include format identifier. The order of the message nodes in the header is designed to support the layered structure by placing the information relevant for the lower layers in the beginning of the compound message.

```

Message ::= Header DataNode

```

4.3 Message data nodes

This is the *payload* of the message thus the information the sender wants sent to the receiver. There are possibilities to combine several messages sent by the same sender to the same receiver (same URN and selectors in both ends).

```
DataNode ::= SingleMessage | Multipart.
```

```
SingleMessage ::= 'd' ';' Length ';' Byte*.
```

The payload is a sequence of bytes representing application-specific data.

```
Multipart ::= '+' ';' Length ';' DataNode*.
```

The payload is a sequence of bytes with the specified length, containing zero or more data nodes.

Examples:

```
d;5;Hello = Hello
```

```
+;18;d;5;Hellod;5;Again = Hello & Again (18 is the length of the underlined part)
```

```
+;40;+;27;d;5;Hellod;5;Hellod;5;Againd;4;Last = ((Hello & Hello & Again) & Last)
```

4.4 Message header nodes

```
Header ::= Version? RoutingR RoutingS Mark* Connection Reliable? Chopped?.
```

Header nodes *Version*, *Mark*, *Reliable* and *Chopped* are optional. *Mark* can be present zero or more times.

Version. *Version* info is only sent together with certain messages during discovery. Nodes not using an acknowledged version are ignored. Details of how to represent versions are still to be defined.

```
Version ::= 'v' ';' Length ';' VersionIdentifier.
```

Mark. Additional information about a message.

```
Mark ::= 'm' ';' Length ';' Byte*.
```

A sender can add arbitrary information to a message using this header node. Markers can for example be used by a routing device to tag a message in order to detect a message that is traveling in a loop in the network. It is up to the user of a mark to define the structure and contents of the data-part of the mark.

There can be more than one marker in a message, for example put there by different routers the message has passed through.

Details on the rest of the header nodes can be found in the following sections.

5 Communication Model Layers

5.1 Media Abstraction Layer

The Media Abstraction Layer (MAL) provides the low-level components of the communication model. These are protocol-specific components that encapsulate driver-specific information and provide a standard API to the upper layers. That API is encapsulated in a `MediaManager` as shown on Figure 2.

```
public void send(Message message,
                URN senderURN, Selector senderSelector,
                URN receiverURN, Selector receiverSelector);
```

Send a message to a given device at a given selector.

```
public void sendBroadcast(Message message, URN senderURN,
                          Selector senderSelector);
```

Send a broadcast message.

```
public Selector startReceiving(PalcomThread receiver);
```

Start receiving messages at a selector chosen by the media manager.

```
public void startReceivingBroadcast(PalcomThread receiver);
```

Start receiving broadcast messages.

```
public void stopReceiving(Selector selector);
```

Stop receiving messages at the given selector

```
public void stopReceiving(PalcomThread receiver);
```

Stop receiving messages by the given receiver.

```
public void addErrorHandler(PalcomThread handler);
```

Subscribe to error events from this media manager

Figure 2: Media Manager.

Currently, media manager implementations exist for UDP/IP on PAL-VM and JVM, Bluetooth on JVM and Infrared on PAL-VM.

5.2 RASCAL Layer

RASCAL (Resilience and Adaptivity System for Connectivity over Ad-hoc Links) provides a layer of routing intelligence that is, at this stage of the project, entirely optional because, due to its dependency on the JADE [1] middleware, it can only run on the JVM. The layer provides to the higher stack layers, including the Service Layer, the possibility to abstract over the particular network bearer to be used for sending and receiving information, unless they wish to specify or influence selection directly.

The *connection-aware* aspect of RASCAL allows the expression of policies to be enacted by this layer in accordance with network status when preparing to send or receive messages. Policies can be based on the bearer network technologies available or on some QoS parameters such as the load of the nodes/devices involved in the

communication or with contingency situations such as failovers. A connection-aware policy will consider using an alternative interface for sending messages to a particular node if a currently employed interface is unavailable.

The *usage-aware* aspect of RASCAL allows the expression of policies to be enacted by the RASCAL control agent in accordance with deployed user-level services. Examples of these are: contingency management decisions, content adaptation decisions, deferred service provisioning decisions, role management decisions, etc. An example of contingency policy particularly useful in disruptive environments consists of sending the message using two or more different routes simultaneously to maximize the potential for a message to reach a target node. All policies allow the RASCAL Layer to dynamically adapt communication behaviour according to particular scenarios in order to exhibit stability to the PalCom user.

RASCAL does not, however, support per-hop routing decisions as this is expected to be the ultimate responsibility of the Routing Manager.

RASCAL Architecture As mentioned, currently the RASCAL Layer is optional and need only be included into the PalCom communication stack if autonomic management of available bearers is required. A PalCom Node with the RASCAL communication layer activated is colloquially termed as being 'Rascalized'.

The RASCAL layer is built using JADE [1], a software agent middleware. The operational logic of RASCAL is thus managed by a software agent. One of the primary design features of RASCAL is its integration with the PalCom communication stack used by all PalCom devices. Within this stack RASCAL presents itself as a Media Manager (MM) following the *decorator* design pattern [3]. This allows RASCAL to be easily plugged, transparently intercept and treat the service information flowing through a PalCom node between network connection endpoints. The physical media managers (e.g. UDP, Bluetooth, IR, etc.) are dynamically loaded by the RASCAL layer through reflection.

One of the goals of RASCAL is to provide the end user with an easy means of writing network or usage aware policies. Thus policies are not coded directly within the agent behaviours but rather with a specialized open source policy server called Ponder2 [2] which provides both a policy engine and a XML human readable policy description language.

RASCAL API The use of the RASCAL Layer is straightforward and can be performed easily modifying a line of the device code (see below). Its API is the same provided by a generic PalCom media manager.

```
public MediaManager getMediaManager() {
    if (mediaManager == null) {
        mediaManager = new RascalMediaManager(getScheduler());
    }
    return mediaManager;
}
```

Further details of the RASCAL Layer of the Communication Model are available in *Deliverable 44* [12] and *Deliverable 45* [13].

5.3 Routing Layer

For the Communication layer of the PalCom communication model, there is a requirement of support for routing communication between devices that are on different networks. The networks may be of different topologies, and using different network technologies.

The general challenge is as follows: given two devices, A and B, who do not share any network technology or protocol, how do we enable communication between A and B, using one (or more) intermediate device(s) - X_k ?

The simple case, illustrated in Figure 3, could be exemplified by the following example:



Figure 3: The general version of the routing challenge

Some service running on device A needs to make use of functionality solicited by another service, running on device B. Device A, however, is only able to speak WiFi and as device B is only able to speak Bluetooth, they are effectively unable to speak directly with each other. There is, however a device, X_1 , reachable by both A and B, able to speak both WiFi and Bluetooth. How do we enable X_1 to route communication between A and B?

Different approaches for solving this, and challenges associated with those approaches, are discussed in more detail in Section 6.2.

There are preliminary implementation of the DSDV [4] and SMURF routing protocols. They take the form of a routing manager that has the same API as a media manager (Figure 2). This is very much work in progress.

5.4 Communication Layer

This layer implements the communication mechanism which takes care of queueing incoming messages to the corresponding listener. It also facilitates reliable connections and handling of messages larger than the maximum message size of the underlying network.

The basics of the Communication Layer are communication components that are abstractions on top of some networking protocols, for instance TCP/IP, UDP/IP or Bluetooth, depending on the capabilities of the specific devices on which they are implemented.

By providing this abstraction, the higher-level APIs becomes independent from the specific network hardware and protocol being used, thereby providing a uniform access regardless of the underlying network.

The basic communication model of PalCom defines, but is not restricted to, two components, one providing *publish/subscribe* communication, and the other—*point-to-point* style communication. The demand for publish/subscribe comes from the facilities needed for the implementation of event notification interactions, in particular the discovery protocol. The demand for point-to-point comes from the highly dynamic nature of palpable systems where message exchange is the preferred method of communication. Additionally, there's support for connection-oriented point-to-point communication if it better fits the requirements of a particular application.

5.4.1 Publish/Subscribe Communication Component

The Publish/Subscribe Communication Component provides a fully distributed scheme for devices and services to communicate, centered around a *publish/subscribe* communication model (based on e.g. broadcast or multicast). This scheme allows for a time, space and flow decoupling between the announcement and discovery of services.

The publish/subscribe capabilities provided by the component are defined to be *probabilistic*. This means that it is very likely, though not guaranteed, that a published message will arrive to the subscribers. In other words, the publisher has no guarantees that the subscribers will receive the published messages.

Part of this probabilistic property stems from the time and space decoupling. From this, the publisher does not know about the presence of the subscribers, if any, at the time when it publishes messages. As such, it cannot

know if any subscribers receive the message. In addition, there are no guarantees about delivery time, that is, the time between a message is published and when it is received by the subscribers. For example, a subscriber *re-appearing* on the network, after having been temporarily unavailable, while a message was published might still be able to receive the message, though this is also not guaranteed. As such, the semantics of the subscriber is that of *non-persistent*.

In addition, the probabilistic property means that messages to a slow subscriber, that is, one that processes the messages much slower than they are received whereby the messages queue up, might be discarded by the PalCom runtime environment to avoid memory problems.

On the wire, publish/subscribe messages are structured as multi-part messages (of type +), with the topic as the first part, and the actual message as the second part. The messages are sent to a broadcast or multicast channel, which means that the selectors in the message are not used, and can be set to zero. The data sent out for a publish/subscribe message will be as follows, assuming that the length of the data in the actual message is the string `mymessage` (9 bytes), and the topic string is `mytopic` (7 bytes):

```
+;24;d;7;mytopicd;9;mymessage
```

This is the wire format of the message, byte by byte. Note that in this case all bytes are printable characters, including the message data, so the wire format can be shown this way.

The basic broadcast or multicast based publish/subscribe communication scheme is accessible to the services through a simple API, see Figure 4. The API defines methods for publishing messages as well as subscribing and unsubscribing receiver threads to the reception of given messages. When a message is published that matches the topic specified by a subscriber, the subscriber is notified about its arrival by sending an event to the provided thread.

```
public void publish(String topic, Message msg);
Send a message to all subscribers to the given topic.

public void subscribe(String topic, PalcomThread subscriber);
Subscribe to receive messages of the given topic.

public void unsubscribe(String topic, PalcomThread subscriber);
Unsubscribe from the given topic.
```

Figure 4: API for publish/subscribe

5.4.2 Point-to-Point Communication Component

The Point-to-Point Communication component provides the ability for the services to communicate in a connection-oriented and connection-less, point-to-point scheme. The component basically acts as an abstraction on top of a low-level communication protocol in the Media Abstraction Layer, using the wire message format described in Section 4.

For message passing, the Point-to-Point Communication component has the API shown in Figure 5.

The `send` method sends a message to the receiver address in the message (the address is in the format described in Section 3). `startReceiving` starts receiving messages for a thread (a service), and returns an address to

```

public void send(Message message, URL receiverURL);
Send a message to the specified receiver.

public URL startReceiving(PalcomThread receiver);
Start listening for incoming messages.

public void stopReceiving(URL url, PalcomThread receiver);
Stop listening for incoming messages.

```

Figure 5: Point-to-point API.

use for sending messages to it (this URL is typically distributed to other services via the discovery protocol). `stopReceiving` can be used for stopping to receive messages at a URL.

The point-to-point communication component provides asynchronous sending and receiving (all methods are non-blocking). If synchronous calls are needed by the services, a higher level component must be created to provide such an abstraction.

Functionality in the Media Abstraction Layer is utilized for handling maximum transmission units, and other issues specific to the underlying network technology.

Message broadcast

Messages destined for broadcast are marked with the `Broadcasted` header:

```
Broadcasted ::= 'b' ';' Selector.
```

These are messages sent to all nodes/URNs. The selector is used to deliver the message to the specific thread implementing a service on the node. For discovery broadcasts, the selector is always 1. For these broadcasts (*Selector* = 1) replies are broadcasted to the same selector. For broadcasting services, the sender uses the selector in the `ServiceDescription` of its service and thus the listener needs to use the same selector to listen in. For these broadcasts there are no replies. Note: For broadcasted messages there is no `RoutingR` header node.

Establishing Connections

In many of the PalCom scenarios, there are situations where two services exchange a sequence of messages. In order to support this common case, the Point-to-Point Communication component provides a connection abstraction. These connections build upon the message sending described above, and give the same degree of guarantees for message delivery. See Section 6.3 for a discussion about specification of quality of service for message sending.

In the API (Figure 7), a connection is presented in a symmetrical way to both parties. The connection object fills in sender and receiver addresses in all messages sent, and handles shut-down when one party closes the connection.

For the purpose of connection management there's the `Connection` message header:

```
Connection ::= 'c' ';' Length ';' ConnectionTag.
```

The different types of `Connection` messages are distinguished by their `ConnectionTag`.

`Open` – establish a connection.

```
Open ::= 'o' ';' Selector ';' Selector.
```

The first selector is the one found in the `ServiceDescription` for the service. The second selector is the one used by the sender.

`OpenReply` – confirm connection. The connection layer locally allocates a unique new selector used for the new connection (listened to by a thread implementing the `Service` in question) and responds to the request by replying to the sender with the following `ConnectionTag`:

```
OpenReply ::= 'p' ';' Selector ';' Selector.
```

The first selector is that of the sender of the `Open` request. The second is the new selector that will be used by this side of the connection. On receiving an `OpenReply` message the customer makes sure that its connection associated with the selector on the sending side is using the new selector at this side for further communication.

`Close` – terminate a connection.

```
Close ::= 'c' ';' Selector.
```

The `Connection` associated with the selector is removed and the associated thread is informed that the `Connection` is *closed*.

`Re-open` – the connection is no longer available.

```
Re-open ::= 'r' ';' Selector.
```

This is a response to `MessageOverConnection` or `SingleShotMessage`. The connection that used to be associated with the selector is no longer available (the device probably re-booted). Please connect again to get a valid new receiver selector.

`MessageOverConnection` – send data over a connection.

```
MessageOverConnection ::= 'm' ';' Selector.
```

For an incoming message the selector is used to identify the corresponding `Connection` (resulting from an earlier `OpenReply` message) and associated service-level thread. Correspondingly, an outgoing message, over a `Connection`, is tagged as `MessageOverConnection` with the selector of the receiving node (as the sender sees it).

`SingleShotMessage` – message without payload.

```
SingleShotMessage ::= 's' ';' Selector ';' Selector.
```

The first selector is that of the receiver, the second–of the sender. An incoming single-shot message is using the receiver selector of a service and the message is routed to the associated thread. The URN and selector of the sender are passed along to the listener for use in event of a reply to the sender. When sending a single-shot message the sender is providing the URN and selector of the receiver, getting them either from an announced service, or from the original message, which it replies to.

Where there's the need for a reliable connection, i.e. one that guarantees the reception of all messages in the order in which they are sent, the message header contains additional tags:

```
Reliable ::= 'R' ';' Length ';' Seq.
```

Seq - a sequence number over the reliable channel, taking the values: $1, 2, \dots, 2^{16} - 1, 0, 1, \dots$

If a message with a matching sequence number is received (i.e., $Seq = PreviousSeq + 1$), an acknowledge message is sent back to the sender, over the same connection, indicating that the message has safely arrived.

```
AckMessage ::= 'A' ';' Length ';' Seq.
```

If sequence numbers do not match, one or more requests for re-sending is sent (with a sequence number set to $PreviousSeq + 1, +2 \dots, Seq$).

```
ResendMessage ::= 'B' ';' Length ';' Seq.
```

The Communication Layer makes sure messages arrive to the listener in order, possibly by buffering later messages if messages are dropped.

On the other hand, if the original sender does not receive an Ack or ResendMessage within some timeframe it must resend the oldest not acknowledged message.

Reliable communication in a connection is initiated by the sender by including a Reliable node. The first time a receiver detects a Reliable node (with $Seq = 1$) it takes this to mean that the sender wants to switch to reliable communication.

Large messages are sent in pieces in messages that have the optional Chopped tag.

```
Chopped ::= '-' ';' Length ';' Number ';' Number.
```

The first number is part number, where 1 means the first part of a chopped long message. The second number specifies the total number of parts. If the two are equal this is the last part of the chopped message.

When the layer receives such a message the parts are assembled in order. If parts are missing or arrive out-of-order the whole partial message is dropped. Note that a reliable chopped connection is possible in which case no such problems arise.

For connections, the API is as shown in Figure 6.

Here, `open` opens a connection to a remote address specified with the URN of the remote service and a selector. When the connection has been established, the specified connection listener will be notified about the new connection, and given a Connection object to use (see below). On the opposite side, `startListening` can be used for starting to listen for connections, and new connections will be reported to the listener in a similar way as for `open`. `startListening` returns the URL that can be used for connecting. `stopListening` stops listening at a URL.

Connection objects implement the interface shown in Figure 7, which is the same at both sides of a connection.

The `send` method sends a message to the other side of the connection. There are two methods for adding and removing message receivers—`addMessageReceiver` and `removeMessageReceiver`. The two `add/removeDisconnectReceiver` methods are used for getting a notification when a connection has been closed.

Appendix C gives a small example of code that uses these connections.

```
public void open(URN localURN, URN remoteURN, Selector remoteSelector,
                PalcomThread connectionReceiver);
Initiate a connection with a remote service.

public URL startListening(PalcomThread connectionReceiver);
Start Listening for incoming connection requests.

public void stopListening(URL connectionURL);
public void stopListening(Selector selector);
Stop listening for connection requests.
```

Figure 6: API for connections.

```
public void send(Message message);
Send a message over this connection.

public void addMessageReceiver(PalcomThread receiver);
Add a listener for messages received over this connection.

public void removeMessageReceiver(PalcomThread receiver);
Remove a listener for messages received over this connection.

public void addDisconnectReceiver(PalcomThread receiver);
Add a listener for the termination of this connection.

public void removeDisconnecReceiver(PalcomThread receiver);
Remove a listener for the termination of this connection.
```

Figure 7: Interface of Connection objects.

5.5 Function Layer

The function layer, located on top of the communication layer, defines, but is not restricted to, two function APIs:

- *Discovery* – used for the announcement and discovery of services on the network.
- *Data Streaming* – typically used by services for transmitting open-ended information such as audio or video.

These components provide an easy to use API for the services to use. However, as abstractions they are just built on top of the lower level components and can be skipped, if needed.

5.5.1 Discovery

Before two or more services can collaborate with each other by communicating, they have to be aware of each other's presence on the network. Announcement and discovery enable the mutual awareness of services and the devices on which they are hosted, on a given network at a given time.

The discovery protocol described here has been developed in close contact with the work on End-user composition (WP6) and has been inspired by the demands from the application prototypes. From these scenarios one can conclude some initial design goals, most importantly that there should be no central coordinator. This has led us to propose a protocol that is broadcasting messages to all devices on the available network, using publish/subscribe. Such “broadcast” protocols are known to overflow large networks if used naively so we propose a protocol that use broadcasting as carefully as possible. As the PalCom architecture represents different technologies as components in the Media Abstraction layer, we are able to tailor the broadcast strategy chosen. The scope of this suggestion is for local networks. Large and/or heterogeneous networks are discussed under Section 6.

For a review of state-of-the-art within the area of discovery protocols, and the issues of device and service discovery related to the six PalCom challenges see [10]

Rationale and Expected Use

PalCom devices should be possible to combine in ways that are not pre-planned, in ad-hoc ways. This means that the PalCom discovery protocol must be understood and supported by all PalCom devices in order for them to communicate at this general PalCom level.

In order to be an Open Framework it must also be possible to implement PalCom devices not only on top of the PalCom prototype framework, but also using other technologies for example to support small devices, or add support for PalCom on existing devices. The communication on the discovery level must thus be defined and standardized as a protocol with regards to formats and representation.

The expected direct use of the protocol is to implement PalCom on small devices, too small for housing the PAL-VM. Here it is important that the use of the PalCom protocol in this situation is lightweight in terms of memory usage and processing power. The situation is similar when implementing PalCom on existing devices. Another use of the protocol specification is during low-level debugging of new PalCom devices.

The discovery protocol enables PalCom devices to discover each other, to find out what services they offer, and handling connections between devices. The details of individual services, service descriptions, we see as a separate PalCom protocol and, for the time being it is not documented in this deliverable.

For the service protocols, specified in ServiceDescriptions, there is a standard format for implementing them. It should, however, also be possible to use a specialized message protocol for especially demanding situations.

There are thus three different protocols involved in PalCom communication:

- *The discovery protocol* enables devices to find other devices, services and connections.
- *Service descriptions* describe the capabilities provided by a service.
- *A service protocol* defines the messages and format used by a particular service.

The two first protocols must be supported by all PalCom devices. For the third protocol there is a PalCom Standard Service Protocol, but this is optional for a service to use.

Overview of the Discovery Protocol

In the following sections we will give an overview of how the PalCom discovery protocol works, what kind of messages are being sent and what kind of information they contain. The exact details of how the messages and parameters are represented are given in Appendix B.

Device Discovery Procedure There are several situations when a PalCom device needs to know what other PalCom devices are available. When a device is booted and when it has been moved so the reachable devices have changed. Many devices also need to keep an up-to-date list of reachable devices and thus need to know when other devices come and go.

The device that needs to know what other devices are available broadcasts the following message (the notation *All.message* indicates a broadcasted message):

All.Request-DeviceInfo(Name,Address)

– with the interpretation: “I am ‘name’ at ‘address’ - who are you?”

Receiving this message all other devices broadcast a similar reply:

All.DeviceInfo(Name,Address)

– with the interpretation: “I am ‘name’ at ‘address’”

After this sequence all the Devices have sent one message each and received one message from each other device. They all thus have had the possibility to update their list of available devices, add new devices, and remove devices that no longer answer.

One crucial design decision here is how often this discovery procedure is initiated. If it is done too seldom a user will notice a delay between the time things should be available and when they are actually shown as available. If they are made too often the resulting messages can overflow basically any network. The design chosen here is to initiate the discovery procedure on demand—that is when needed. The most eager device to know about the others initiates the discovery procedure, and all the other devices will be updated as well as a side effect.

The PalCom discovery protocol does not impose a specific strategy for how the discovery mechanisms are used, but this is up to the implementers of PalCom devices. The discovery procedure will typically be initiated by devices running for example a PalCom browser, and thus provide a user interface, which is showing the available devices. A possible strategy for that situation is to send out requests relatively seldom, say if the available list is older than 10 seconds. This will guarantee that the view presented is fairly up to date. If there is another device sending out requests more frequently, the less demanding device will never actually send out its requests. On top of this mechanism the browser could offer an update button, so the user can refresh the view more frequently when needed.

For “small devices” it is important that the overhead inflicted by the discovery protocol is small. Such a device would be one that simply offers a service, it will thus never initiate a discovery procedure. It can answer requests with a simple, often fixed answer and it can ignore the broadcasted replies from the other devices.

Device Inactive Notification A device that is about to be orderly shut down can inform other devices that so is the case. It does so by broadcasting a message:

All.Inactive-DeviceInfo(Name, Address)

– with the meaning: “I, ‘name’ at ‘address’ am about to shut down”

Devices receiving this message should remove the sending device from its list of available devices and disconnect all connections they have with services on that device. Devices can thus be removed from listings on other devices both explicitly, after receiving this Inactive-DeviceInfo notification, or indirectly when a device fails to respond to the Request-DeviceInfo.

Service Listing Procedure A device has discovered another device through the device discovery procedure above. It thus knows the addresses of the available devices. If it (D1) needs to know about what services are available on a specific device (D2), it sends a single message to that device:

D2.Request-ServiceList

– with the meaning: “Please send me a list of your services”

Receiving this message the addressed device (D2) will answer with

D1.ServiceList(S)*

– with the meaning: “My services are: S1, S2, etc”

A device offering services must thus provide a list of its services. Services can be organized hierarchically which is expressed in the representation of them in the parameter S*. The services have a name and an address each. They can thus be addressed as separate entities although they of course reside on a device.

There are different kinds of situations, where services are offered: A “small device” can reply with a fixed, prepared in advance representation of its services making this part of the protocol easy to support. Browsers, and other devices that can activate assemblies, need to represent a dynamic list of services including the services offered by its active assemblies. In the same way providing execution of software components means that the services offered by these must be included in the resulting service list.

A device can request a list of services as often as it needs. A combination of automatic update at a relatively slow pace and a possibility to update on demand is probably a good strategy. Again, the update strategy is not part of the protocol, but up to the device implementers to tailor.

Service Description Before a device can start to use a service on another device it must know what protocol is used for communication with that particular service. These service protocols are described in service descriptions. A device, D1, that wants to know the details of a particular service (S) sends a request (it knows the address of it from a ServiceList reply):

S.Request-ServiceDescription

– with the meaning: “Please send me a description of yourself”

and the service replies to the requester:

D.ServiceDescription(SD)

– with the meaning: “Here is my service protocol, SD”

There are several obvious reasons for requesting a ServiceDescription. A user might want to directly interact with the service and wants to use a browser as a remote control for it. The requested ServiceDescription will in that case be rendered as a user interface by the browser. A user may be putting together an assembly and need to know what messages its protocol can accept and send. An assembly is about to be activated and it is verified that the protocol expected by the assembly is actually provided by the service. The content of a ServiceDescription is such that these examples can be handled provided that the Standard Service Protocol is used.

It is, however, also clear that there are applications with demanding needs, exploring the available future technology with maximum efficiency. In such cases one might need to use handcrafted optimized service protocols, and it should be possible to do that for PalCom services. It is at this point, however, not decided on exactly how they will be specified in a ServiceDescription. This is a trade-off between flexibility in tailoring a specialized protocol, and conformance with PalCom and the support that can be offered. Here the further development will be guided by the application prototypes.

Remote Connect One device can initiate a connection between services on two other devices. It does so by sending the following message to the service S1, the one of the two that plays the role of customer:

S1.RemoteConnect(S2)

– with the meaning: “Please connect to the Service S2”

The need for this functionality appears for example when a browser, executing on one device, is used to connect two services on two other devices. This can be part of a direct user action in the browser or indirect, when an assembly is activated. The mechanisms for actually establishing a connection between two services is part of the API described in Section 5.4.2 about connections.

Remote Disconnect This is the counterpart to Remote Connect with similar motivation. Sending the message to the service playing the role of customer (S1) the following message:

S1.RemoteDisconnect(S2)

– with the meaning: “Please disconnect from the Service S2”

Connection Listing It is possible to ask another device what connections it has initiated (thus where it has a service that play the role of a customer). It does so by sending the device (D2) the following message:

D2.Request-ConnectionListing

– with the meaning: “Please send me a list of connections you have initiated”

The receiving device will answer with:

D1.ConnectionListing(C1(S11,S12), C2(S21,S22), ...)

– with the meaning: “I have connected my service S11 to S12, and so on”

The answer is a list of connections with enough information to re-establish the connection at the later stage. So sending S11 the message RemoteConnect(S12) should re-establish the connection C1, etc.

Discussion

The approach taken in the proposed discovery protocol is based on broadcast without a central. In this section we try to analyze and discuss how it will behave with respect to different design goals and in particular its potential for scaling up in larger networks. Initially we presume a situation with a single network with a limited number of devices. The devices available are those that are visible on that network.

Protocol-inflicted delays It is a design goal that the responsiveness of PalCom is sufficient for interactive work. From a user perspective this can be illustrated with the situation when a user studies the available devices in a PalCom browser. How long will it take before a change in the number of available devices is reflected for the user in the worst case?

There are five cases how the situation can change, where X is one of the devices:

1. Device X boots.
2. Device X is shut down uncleanly.
3. Device X is shut down cleanly.
4. Device X comes within reach.
5. Device X leaves.

In the discovery protocol there will be direct notification to all other devices in case 1) and 3). Here the protocol will thus not inflict any extra delay. In the other three cases the change will be noticed at the next round of the discovery procedure. The time from the change takes place until it is actually reported depends on the time until the most eager device initiates a new discovery procedure. This time is thus determined by the applications and not by the discovery protocol itself. It can for example be set very short for a while by a device that is in a critical state. This can be lifted all the way up to the end user to control.

Protocol-inflicted communication load Another design goal is that the protocol must be efficient from the point of view on putting load on the communication channel. It is a PalCom design choice that there is no central server.

When a new device becomes present (boots or comes within reach) it needs to get information about all other available devices. In an environment with N devices, this requires one initial request and $N - 1$ replies which is what the suggested protocol will use. Since these messages are sent as broadcasts it means that at the same time all other devices are brought up to date as well (including registering the new device). For updating a situation with N devices we require N messages. If single messages (unicast) had been used for answering the request there would have been needed $N * N$ messages. We thus conclude that the proposed protocol is optimal.

Discovery API

For the programmer, there is a class `DiscoveryManager`, which handles discovery with help from lower-level communication components. Its API, in Java, is as shown in Figure 8. The structure of the `ServiceList`, the `ServiceDescription`, and the `ConnectionInfo` follow the structure described in Appendix B about message formats.

5.5.2 Data Streaming

The Data Streaming component is used for sending data, which conceptually does not fit into a single message. One typical example is the streaming of audio or video data, where the processing of the data on the receiving service should take place at reception time, and thus before the final piece of data, if any, has been received. Data streaming is unreliable in the sense that there can be missing information. Technically, pieces of the sent information arrive in order, but without delivery guarantee, that is, pieces of the information arriving out of order are dropped.

The Data Streaming component can be configured to communicate using the point-to-point or the publish/subscribe communication components of the Communication layer. The latter is particularly useful when there are multiple services interested in receiving the stream of data.

5.6 Service Layer

The communication model enables the services to communicate by using the APIs of the communication components in the Function layer. However, if needed the communication model also allows for the services to use the communication components of the Communication layer directly, thus bypassing the components in the Function layer.

6 Open Issues

For the PalCom Communication model, there are several areas where more examination and experimentation are needed. This section discusses a number of these, and points out different possibilities for the future evolution of the model.

```
void initiateDiscovery(long timeout);  
Initiate discovery by sending out the local device info with isRequest=true.
```

```
void updateDeviceInfoReceiver(PalcomThread receiver);  
Add a receiver that will be notified as DeviceInfos are received.
```

```
void removeDeviceInfoReceiver(PalcomThread receiver);  
Remove a receiver
```

```
void updateLocalServiceList(ServiceList list);  
Update a list of local services. The list can be the list of top-level services, or a list of subservices to a service (depending on how the parentURN attribute is set in the list).
```

```
void removeLocalServiceList(ServiceList list);  
Remove a list of local services. The list will no longer be delivered to other devices.
```

```
void updateLocalServiceDescription(ServiceDescription sd);  
Update the description of a local service.
```

```
void removeLocalServiceDescription(ServiceDescription sd);  
Remove a local service description. The description will no longer be delivered to other devices.
```

```
void updateServiceListReceiver(URN deviceURN, PalcomThread receiver);  
Add a receiver that will be notified as ServiceLists are received.
```

```
void removeServiceListReceiver(URN deviceURN, PalcomThread receiver);  
Remove a receiver that will be notified as ServiceLists are received.
```

```
void requestServiceList(String parentURN);  
Send a request for the list of subservices of the device or service identified by parentURN. If the URN refers to a service, the URN of the device to send the request to is extracted from the service URN.
```

```
void requestServiceDescription(String serviceURN);  
Send a request for the description of the service identified by serviceURN. The URN of the device to send the request to is extracted from the service URN.
```

```
void updateServiceDescriptionReceiver(URN serviceURN, PalcomThread receiver);  
Add a receiver that will be notified as ServiceDescriptions are received.
```

```
void removeServiceDescriptionReceiver(URN serviceURN, PalcomThread receiver);  
Remove a receiver that will be notified as ServiceDescriptions are received.
```

```
void updateLocalConnectionInfo(ConnectionInfo ci);  
Update a local connection info.
```

```
void removeLocalConnectionInfo(ConnectionInfo ci);  
Remove a local connection info. The info will no longer be delivered to other devices.
```

```
void requestConnectionList(String deviceURN);  
Send a request for the list of connections initiated by the device identified by deviceURN.
```

```
void addConnectionListListener(InfoListener listener);  
Add a listener that will be notified as ConnectionLists are received.
```

Figure 8: Discovery component API

6.1 Discovery

6.1.1 Architectural Elements to be Discoverable

Section 5.5.1 has presented procedures for discovering devices, services, and connections. One important issue to tackle is to identify additional architectural elements whose descriptions need to be discovered. Some candidates are:

- *Resources*. The preliminary study performed within WP5 [11] has already pointed out the need of a (possibly quite rich) resource description. As soon as the requirement of aggregating resources deployed across a network is considered, it's very likely that some form of resource advertisement is needed.
- *Components*. PalCom components, and in general uninstantiated entities, can themselves be advertised as a dynamic deployment information that would enable hot-swap and live upgrade of subsystems, and most likely preventive resource control (i.e., some checks could be made even before instantiating a resource, and when unfeasibility can be foreseen, the instantiation request can be refused right away).
- *Specific Runtime Components*. Instantiated components with additional semantics such as *Palpable Agents* or *Frugal Objects* could require their own extension of the standard service description.
- *Actors*. It could be envisaged to have advertisement of actors, mostly as principals in an authentication system.

6.1.2 Large Numbers of Discoverable Elements

In large networks the number of discoverable services can potentially become overwhelming for the user. One possible way to help the user in such a situation is to offer some query and filtering mechanism. The specification of such mechanisms are studied in WP5 and reported in Deliverable 24 [11].

More study and discussion on the issue of representation languages is needed, but the general vision is to exploit description extensibility: to have a simpler base description that somehow gives access on demand to a richer description, that will be exploited only for those entities and those requesters that really need it.

6.1.3 Discovery in Complex Networks

The discovery protocol described in this deliverable has primarily been developed for the situation in a local network and to meet high demands on responsiveness and with less restrictions on broadcasted traffic. In a situation with wide area network these design goals change. The demands on responsiveness are relaxed, the available services are less dynamic and the broadcasted traffic must be kept to a minimum. For this situation it might be that the discovery model will need to be extended with some service directory mechanism. One possibility here is to combine such a mechanism with a gateway between a local area network and wide area network, in which case the problem has some connections to the routing issues in the next section.

6.2 Open issues for Routing

The purpose of this section is to illuminate and discuss different approaches for solving the challenge of routing across different network protocols and technologies, introduced in Section 5.3. Through discussions within the application prototype WPs, two general approaches for routing on a PalCom network have emerged. These are briefly covered here.

Service level proxying

In a proxy based approach for communication, the general model is, for an intermediate device, to proxy functionality not directly reachable by some device. In Figure 9, X₁ functions as a proxy for B.

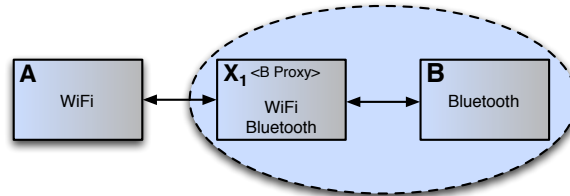


Figure 9: The proxy based approach

This schema is seen extended in Figure 10 to a case with two proxies, X₁ and X₂. In the same way, the schema could be extended to n proxies, ordered not unlike a *chinese box*.

For a proxying strategy, such a service approach would mean that, given some initiation scheme, one or more proxy services are installed on the device. Depending on the level of transparency, this or these services would either imitate having the functionality of- or being a part of, the proxee(s).

Implemented as a service, a proxy could either be instantiated explicitly by users or automatically by the communication component (possibly on discovery time). As a self-contained service, the proxy is interesting with regards to the assembly concept, given that the proxy functionality could be explicitly assembled with the client services. Much of the contingency handling needed could hence be directed upwards toward the user, aligned with common service contingency handling.

For the proxy service approach, a drawback is CPU and memory usage. Speed could also become an issue, given Palcom’s message passing approach. The approach also poses some requirements on the addressing scheme; Having routing/proxying at this high a level, would mean that some other mechanism for mapping addresses to bearers would have to be deployed at a lower level. A value of this approach, is the explicitness and modularity of it (at service level). Smaller devices would simply not own proxy capability. Further, introspection and reflection of the routing mechanism could be more or less free with the service approach.

A general routing layer

A general routing layer is another approach discussed within the prototype WPs. Here, services running on the same VM share a routing layer positioned in the bottom of the overlay network - on top of the various implemented

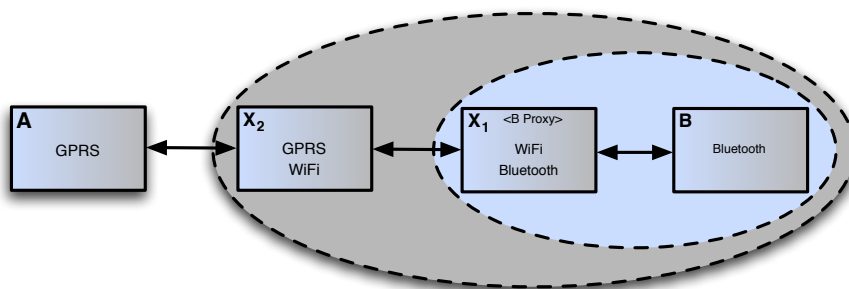


Figure 10: The proxy based approach in the general situation

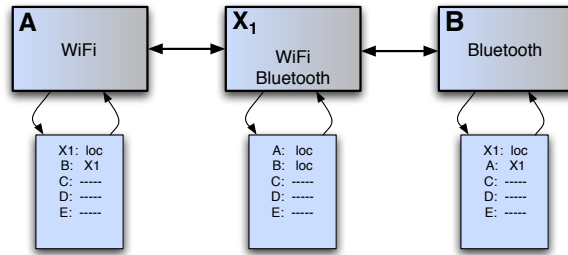


Figure 11: The router based approach

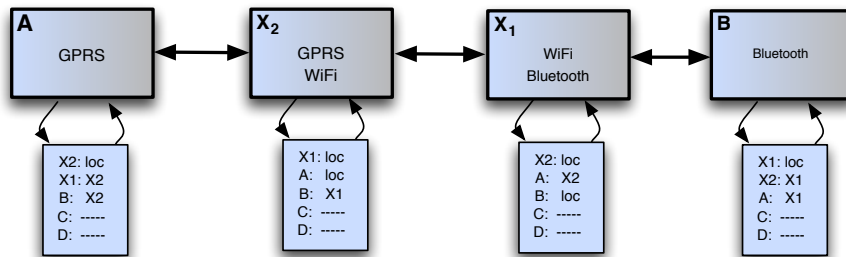


Figure 12: The router based approach in an extended situation.

network technologies. This layer holds some mapping between identifiers of the discovered services the route to them. In figure 11, X_1 routes between A and B.

In figure 12, the schema is extended to two intermediate devices - X_1 and X_2 , routing between A and B. The schema would be extended accordingly.

Given a routing strategy, one could imagine some contingencies being more or less autonomously handled through the router mapping. If this mapping held more than one discovered possible route to the end point, the routing mechanism would be able to route the messages along another path. Contingencies arising from the failure etc. of the end point, should be propagated upwards for handling on a higher level. Given this strategy, it should be possible to scale the routing capabilities of a given service/device according to its resources and needs. This scalability could range from just turning on or off the routing capability, to more fine-grained approaches, where the routing mechanism is customizable at compile- and runtime.

Further Work

Amongst the fields that are currently being investigated are routing in mobile ad-hoc networks and multi-hop routing in peer-to-peer networks. There is also a need to distill further requirements on the routing model. Apart from the prototype WPs, this especially means requirements from WP1, Conceptual framework. These open issues are currently all works-in-progress.

6.3 Quality of Service

In many situations, services may want to set various quality of service parameters of their communication. Examples of this are transactional delivery of messages, or communication over secure channels. This is an area where work is carried out in connection with WP5, resource management.

One possible way to formulate this is as a `QoSPolicy` parameter, grouping several quality of service requirements in a policy, that can be passed to methods of the APIs presented in Figures 4, 5, and 8. It might be that the sender can specify with what quality of service criteria a messages should be sent, or, correspondingly, that the receiver says that only messages fulfilling the provided quality of service criteria should be accepted. This way, it is possible to adjust the default, simple quality of service provided by the component.

Some devices might not support all possible quality of services, for instance the ability to encrypt/decrypt, due to resource considerations. In such a case the communication might proceed with less quality than demanded if that is appropriate, which is up to the service or ultimately the user to decide.

7 Conclusions

7.1 Task 1: Validation of the communication and discovery model and implementation of other core components

The results described in this deliverable show that the communication component has matured significantly. In terms of the APIs that it provides it is available for both virtual machines - JVM and PAL-VM. However, some of the implementation modules are available for either one or the other. In some cases, e.g. the RASCAL layer, the reason is its dependency on software infrastructure only available for the JVM. In others - the lack of support for particular hardware by the runtime platform.

7.2 Task 2: Further improvement of the communication component and improved code-base for the application prototypes

The communication component is central to all PalCom devices. It starts to see some use by the application prototypes. The discovery model has been pretty stable but a transition is now in place that changes the wire format of the messages exchanged during discovery or subsequent communication. The communication component is part of the open source dissemination kit and if it is used that change should be transparent.

Current work is trying to address the goals of the PalCom project in terms of heterogeneous networks by providing, e.g., routing between different network technologies.

7.3 Task 3: Specification of the communication model & other core components

This deliverable presents the specification of the communication model in terms of the addressing schemes, message formats and protocols for discovery and communication. It also presents the specification of the communication component, that is, its APIs. It is hoped to provide sufficient information for the implementors of PalCom system software but also PalCom application software. We hope to see wider adoption of the current implementation by the other workpackages.

References

- [1] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.
- [2] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [4] Guoyou He. Destination-sequenced distance vector (dsv) protocol.
- [5] Internet Engineering Task Force. *Functional Requirements for Uniform Resource Names*, 1994. <http://www.ietf.org/rfc/rfc1737.txt>.
- [6] Internet Engineering Task Force. *Uniform Resource Locators (URL)*, 1994. <http://www.ietf.org/rfc/rfc1738.txt>.
- [7] Internet Engineering Task Force. *URN Syntax*, 1997. <http://www.ietf.org/rfc/rfc2141.txt>.
- [8] PalCom. PalCom External Report 8: Deliverable 6 (2.1): PalCom Open Architecture Overview. Technical report, PalCom Project IST-002057, December 2004. [http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-6-\[2.1\]-Report-on-Architecture-Requirements-And-Design.pdf](http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-6-[2.1]-Report-on-Architecture-Requirements-And-Design.pdf).
- [9] PalCom. PalCom External Report 8C: Deliverable 6 (2.1) Annex C: PalCom Communication Model. Technical report, PalCom Project IST-002057, December 2004. [http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-6-\[2.1\]-Report-on-Architecture-Requirements-And-Design.pdf](http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-6-[2.1]-Report-on-Architecture-Requirements-And-Design.pdf).
- [10] PalCom. PalCom External Report 29: Deliverable 23 (2.4.2): Specification of Component & Communication model. Technical report, PalCom Project IST-002057, October 2005. [http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-23-\[2.4.2\]-component-communication-model.pdf](http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-23-[2.4.2]-component-communication-model.pdf).
- [11] PalCom. PalCom External Report 32: Deliverable 24 (2.5.1): Design Issues for Resource Awareness and Management. Technical report, PalCom Project IST-002057, October 2005. [http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-24-\[2.5.1\]-resource-awareness-and-management.pdf](http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-24-[2.5.1]-resource-awareness-and-management.pdf).
- [12] PalCom. PalCom External Report 58: Deliverable 44 (2.7.2): WP 7-12 Prototypes status after Year 3. Technical report, PalCom Project IST-002057, January 2007. [http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-44-\[2.7.2\]-prototype-status-after-year3.pdf](http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-44-[2.7.2]-prototype-status-after-year3.pdf).
- [13] PalCom. PalCom External Report 60: Deliverable 45 (2.13.2): Fieldwork. Technical report, PalCom Project IST-002057, January 2007. [http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-45-\[2.13.2\]-fieldwork.pdf](http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-45-[2.13.2]-fieldwork.pdf).

A Complete message format specification

The basic constituents of messages are message nodes with the following general syntax:

```

MessageNode ::= Descriptor Data.
Descriptor ::= Format ';' Length ';'.
Format ::= Byte.
Length ::= Number.
Number ::= Digit Digit*.
Data ::= Byte*.

```

Messages are built with specific nodes defined as follows:

```

Message ::= Header DataNode.

```

```

DataNode ::= SingleMessage | Multipart.
SingleMessage ::= 'd' ';' Length ';' Byte*.
Multipart ::= '+' ';' Length ';' DataNode*.

```

```

Header ::= Version? RoutingR RoutingS Mark* Connection Reliable? Chopped?.
Version ::= 'v' ';' Length ';' VersionIdentifier.
RoutingR ::= 'r' ';' Length ';' URN.
RoutingS ::= 's' ';' Length ';' URN.
Mark ::= 'm' ';' Length ';' Byte*.
Connection ::= 'c' ';' Length ';' ConnectionTag.

```

```

ConnectionTag ::= Broadcasted | Open | OpenReply | Close | Reopen
| MessageOverConnection | SingleShotMessage.

```

```

Broadcasted ::= 'b' ';' Selector.
Open ::= 'o' ';' Selector ';' Selector.
OpenReply ::= 'p' ';' Selector ';' Selector.
Close ::= 'c' ';' Selector.
Reopen ::= 'r' ';' Selector.
MessageOverConnection ::= 'm' ';' Selector.
SingleShotMessage ::= 's' ';' Selector ';' Selector.
Selector ::= Number.

```

```

Reliable = 'R' ';' Length ';' Seq.
AckMessage ::= 'A' ';' Length ';' Seq.
ResendMessage ::= 'B' ';' Length ';' Seq.
Seq ::= Number.

```

```

Chopped ::= '-' ';' Length ';' Number ';' Number.

```

B Message Formats for PalCom Discovery Messages

B.1 Request messages

In the discovery protocol, there are a number of request messages, for requesting information about different entities:

- Requests for information about devices are broadcasted out, together with information about the local device, as discussed in Section 5.5.1. This is done by sending out a DeviceInfo with the isRequest attribute set to true. See Section B.2 below for a description of the wire format of DeviceInfos.
- Lists of services on a device are requested by sending ServiceListRequests to the device. Like all request messages, except requests for DeviceInfos, these are unicast messages. See Section B.3 for a description of the wire format.
- The description of a service can be requested by sending a ServiceDescriptionRequest to the device the service is on. See Section B.5.
- A list of the connections that a device has initiated can be retrieved by sending a ConnectionListRequest to the device. See Section B.7.
- For requesting a customer to connect to a provider, a RemoteConnect can be sent to it. See Section B.9.
- Correspondingly, for requesting a customer to disconnect from a provider, a RemoteDisconnect can be sent to it. See Section B.10.

B.2 DeviceInfo

In the discovery protocol, the only information that is distributed in a broadcast fashion is the information about devices in DeviceInfos. Further information about a specific device can then be retrieved by exchange of unicast messages. All infos are transmitted as XML, as children of an InfoEvent element.

For the broadcast communication, the discovery component uses the publish/subscribe communication offered by the Publish/Subscribe Communication Component, described in Section 5.4.1. Messages are published on the topic "discovery", and interested devices receive the messages by subscribing to that topic. When looking at the wire format used for sending out DeviceInfos, we therefore need to look also at the wire format of publish/subscribe messages.

Publish/subscribe messages are structured as multi-part messages (of type +), with the topic as the first part, and the actual message as the second part. The messages are sent to a broadcast channel, which means that the selectors in the message are not used, and can be set to zero. When communicating in an IP network, the publish/subscribe component uses an IP multicast address and port for the multicast communication¹. For other types of networks, other mechanisms will be used. In the IP case, the contents of the UDP packet sent out for a discovery message will be as follows, assuming that the length of the XML data in the message is 257 bytes:

```
+;276;d;9;discoveryd;257;xml data
```

This is the wire format of the message, byte by byte. Note that all bytes are printable characters, including the XML data, so the wire format can be shown this way. With an example of actual XML data, the message may look like this (line-breaks have been added for readability):

¹Currently, the address 224.0.1.20 and the port 8031 is used.

```
+;276;d;9;discoveryd;257;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE InfoEvent SYSTEM "palcom.dtd">
<InfoEvent keepInfo="true" isRequest="false">
  <DeviceInfo
    urn="urn:palcom://mydevice"
    selector="1"
    name="My device"
  />
</InfoEvent>
```

The outer InfoEvent element is used in all discovery messages (broadcast and unicast). It has two boolean attributes:

- *keepInfo* (required). When keepInfo is true, the info is about an entity that is up and running. When it is false, the entity has disappeared or is about to disappear (and can be removed from lists). When a device is about to shut down, it sends out an InfoEvent with keepinfo set to false.
- *isRequest* (optional). The isRequest attribute is used by the DeviceInfo distribution protocol, as described above: when isRequest is true, the InfoEvent is to be taken as a request for infos about other devices. isRequest can be omitted, which means that it is false.

The actual DeviceInfo has four attributes:

- *urn* (required). The URN of the device (a globally unique identifier).
- *selector*. Selector for further requests. *name* (optional). The name of the device (can be displayed to the user).

B.3 ServiceListRequest

Devices and services that are interested in information about the services on a device can request that by sending a ServiceListRequest to the device. The ServiceListRequest messages are not sent in multi-part messages, like the publish/subscribe messages, but in simple messages. They are unicast, so the selectors in the messages are used.

The following example shows the wire format of a ServiceListRequest message, requesting a list of the services on the device described by the DeviceInfo shown in Section B.2. If the communication is over UDP, this is the contents of one UDP packet (line-breaks have been added for readability):

```
d;147;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE ServiceListRequest SYSTEM "palcom.dtd">
<ServiceListRequest parentURN="urn:palcom://mydevice" />
```

It can be seen that the length of the XML data is 151 bytes. The sender selector of the message is 2, which is the selector used for sending out the request. The receiver selector is 1, the selector the device is listening for requests on.

In the ServiceListRequest, the *parentURN* attribute contains the URN of the device or service for which the service list is requested. In this example, parentURN refers to the device itself.

B.4 ServiceList

Replies to ServiceListRequests come as ServiceLists, in unicast messages. A ServiceList includes a number of ServiceInfo elements, each with information about one service. The services on a device can be arranged hierarchically, in a tree structure: there is one ServiceList for the top-level services on a device, and one for the immediate children of each non-leaf service. Each ServiceList is marked with the URN of the device or service whose children it contains.

The following example shows the wire format of a ServiceList message, that contains the list of services on the device described by the DeviceInfo shown in Section B.2 (line-breaks have been added for readability):

```
d;364;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE InfoEvent SYSTEM "palcom.dtd">
<InfoEvent keepInfo="true">
  <ServiceList parentURN="urn:palcom://mydevice">
    <ServiceInfo
      name="Service 1"
      contentType="application/x-palcom-control+xml"
      role="provider"
      urn="service1"
      selector="11"
    />
    <ServiceInfo
      name="Service 2"
      contentType="image/jpeg"
      role="customer"
      urn="service2"
      selector="12"
    />
  </ServiceList>
</InfoEvent>
```

The sender selector of the message is 11, the same selector as the device is listening for requests on. The receiver selector is 12, which was the selector used for sending out the request. The outer InfoEvent element has keepInfo set to true, which means that the service is still up, and should not be removed from lists.

In the ServiceList, the *parentURN* attribute contains the URN of the device or service to which the service list belongs. In this example, parentURN refers to the device, and the ServiceList contains two services. Each ServiceInfo has the following attributes:

- *urn* (required). This is the URN of the service, relative to parentURN.
- *name* (optional). This is the name of the service (can be displayed to the user).
- *contentType* (optional). contentType is the MIME type handled by the service. The first service in the example handles *application/x-palcom-control+xml*, which is a special MIME type for commands sent to and from a service (see Section B.6 about ServiceDescription). All other MIME types are treated as for the second service: this one handles *image/jpeg*, which means that it can receive JPEG images (receive because it is a customer, see below). For services that are only for grouping of other services, and cannot be connected to themselves, the contentType may be omitted.

- *role* (optional). This is the role the service plays: provider or customer. Provider means that the service can be interacted with through commands (in the *application/x-palcom-control+xml* case), or that it sends out messages with data of the particular type (for other MIME types). Customer means that it can interact with a service through commands, or that it receives data of the particular type. For pure grouping services (see previous bullet), the role attribute can be omitted.

B.5 ServiceDescriptionRequest

Services that handle the MIME type *application/x-palcom-control+xml* can be interacted with using commands. The set of commands available for such a service, parameters of the commands, and groupings of them, are specified in a ServiceDescription. The ServiceDescription can be requested from the device, using a ServiceDescriptionRequest, in a similar way as to how lists of services are requested using ServiceListRequests. Both these requests are sent to the device on which the service is. The wire format of a ServiceDescriptionRequest is as in the following example:

```
d;179;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE ServiceDescriptionRequest SYSTEM "palcom.dtd">
<ServiceDescriptionRequest />
```

B.6 ServiceDescription

The following is the wire format of the ServiceDescription for service1, as requested by the request in the previous section, with its surrounding InfoEvent and message headers:

```
d;364;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE InfoEvent SYSTEM "palcom.dtd">
<InfoEvent keepInfo="true">
  <ServiceDescription
    id="service1"
    providerURN="urn:palcom://mydevice/service1">
    <GroupInfo>
      <Command id="command1" direction="in" />
      <Command id="command2" direction="out">
        <Param id="param1" direction="out" type="text/plain" />
      </Command>
    </GroupInfo>
  </ServiceDescription>
</InfoEvent>
```

The exact structure of ServiceDescriptions has not yet been decided on in detail. The different elements and attributes in the example can be explained as follows:

- The id attributes are used for identifying elements within the ServiceDescription.
- The providerURN attribute is the URN of the service that the ServiceDescription describes.
- Groups are used for grouping commands (can also contain other groups hierarchically).

- Each command has a direction: "in" is used for commands that are sent to the service, "out" for commands that the service invokes itself.
- Commands can also have zero or more parameters. These are used for sending data when invoking a command.

B.7 ConnectionListRequest

Certain applications, such as PalCom browsers, are typically interested in what connections are established between services in the vicinity. Therefore, a device can be asked about what connections it has established. This is done by sending a ConnectionListRequest to the device. The ConnectionListRequest has no attributes:

```
d;93;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE ConnectionListRequest SYSTEM "palcom.dtd">
<ConnectionListRequest />
```

B.8 ConnectionList

In reply to ConnectionListRequests, a device sends a ConnectionList. The wire format is as in the following example:

```
d;200;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE InfoEvent SYSTEM "palcom.dtd">
<?xml version="1.0" encoding="iso-8859-1"?>
<InfoEvent keepInfo="true">
  <ConnectionList deviceURN="urn:palcom://mydevice">
    <ConnectionInfo
      providerURN="urn:palcom://otherdevice/someservice"
      customerURN="urn:palcom://mydevice/service2"
    />
  </ConnectionList>
</InfoEvent>
```

The ConnectionList has one attribute deviceURN, saying what device it belongs to. It has one inner ConnectionInfo element for each connection that the customer has established. The attributes of the ConnectionInfo are the following:

- *providerURN* (required). This is the URN of the provider (as shown in its ServiceInfo).
- *customerURN* (required). This is the URN of the customer.

B.9 RemoteConnect

A browser may send a request to a customer that the customer should connect to a provider. This is sent to the URL of the customer:

```
d;220;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE RemoteConnect SYSTEM "palcom.dtd">
<RemoteConnect>
  <ConnectionInfo
    providerURN="urn:palcom://otherdevice/someservice"
    customerURN="urn:palcom://mydevice/service2"
  />
</RemoteConnect>
```

The RemoteConnect element has an inner ConnectionInfo element, saying what connection should be established.

B.10 RemoteDisconnect

In a way similar to the RemoteConnect, a request for shutting down a connection can be sent to a customer. This is also sent to the URL of the customer:

```
d;220;
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE RemoteDisconnect SYSTEM "palcom.dtd">
<RemoteDisconnect>
  <ConnectionInfo
    providerURN="urn:palcom://otherdevice/someservice"
    customerURN="urn:palcom://mydevice/service2"
  />
</RemoteDisconnect>
```

C Code example with connections

Section 5.4.2 presents a connection API built on top of message passing. The following code is a small example in Java of usage of connections at the provider side:

```
PalcomThread providerSideThread = new PalcomPriorityThread() {
    public void run() {
        while (true) {
            ConnectionEvent event = waitEvent();
            if (event.isConnected()) {
                Connection c = event.getConnection();
                // register the thread that will receive the messages
                // at the provider side
                c.addMessageReceiver(providerReceiverThread);
                c.send(new SimpleMessage("Hello customer".getBytes()));
            } else {
                // disconnected
            }
        }
    }
};
```

```
URL connectionURL = connectionsManager.startListening(providerSideThread);
```

After starting to listen for connections like this, the provider distributes its URN (`providerURN`) and selector (`providerSelector`) to customers through the discovery protocol. A minimal customer may use these data like this:

```
final URN customerURN = new URN("urn:palcom://customer");
```

```
PalcomThread customerSideThread = new PalcomPriorityThread() {
    public void run() {
        while (true) {
            ConnectionEvent event = waitEvent();
            if (event.isConnected()) {
                Connection c = event.getConnection();
                // register the thread that will receive the messages
                // at the customer side
                c.addMessageReceiver(customerReceiverThread);
            } else {
                // disconnected
            }
        }
    }
};
```

```
connectionsManager.open(customerURN, providerURN, providerSelector,
    customerSideThread);
```