

Nokia Research Center / Helsinki
Dirk Trossen, Dana PavelFINAL
24.10.06**N-RSA COMMUNICATION PROTOCOLS & REMOTE INTERFACE APPLICATION
PROGRAMMER INTERFACES**

Authors: Dirk Trossen, Dana Pavel
Nokia Research Center / Helsinki

Editor: Elena Balandina
Nokia Research Center / Helsinki

ABSTRACT

This document outlines the protocols and application programming interfaces that are used with the Nokia Remote Sensing Architecture (N-RSA) for remote communication between the application server and the set of mobile gateways. Note that the local sensor communication is not within the scope of this document. This local communication is implemented by the specific gateway handler for the particular sensor network.

Version:	1.0
Authors:	Dirk Trossen, Dana Pavel

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

GLOSSARY

AM	Application Module
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JNI	Java Native Interface
MTU	Maximum Transfer Unit
N-RSA	Nokia Remote Sensing Architecture
OTA	Over-the-Air
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XCAP	XML Configuration Access Protocol
XML	Extensible Markup Language

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

TABLE OF CONTENTS

1. INTRODUCTION..... 5

2. EVENT DELIVERY PROTOCOL FRAMEWORK..... 5

2.1 PROTOCOL DATA STRUCTURES 5

2.2 ENDPOINT IDENTIFIERS..... 8

2.3 PROTOCOL OPERATIONS..... 8

 2.3.1 *Subscription & Notification*..... 8

 2.3.2 *Termination of Subscription*..... 10

 2.3.3 *Publication*..... 11

2.4 TEMPLATE FOR EVENT DELIVERY MAPPING..... 12

 2.4.1 *Identifiers*..... 12

 2.4.2 *Definition of Transport Service*..... 12

 2.4.3 *Marshalling of Data Structures*..... 13

 2.4.4 *Handling of Inbound Traffic*..... 13

 2.4.5 *Authentication and Encryption*..... 13

3. EVENT DELIVERY MAPPINGS..... 13

3.1 TCP MAPPING..... 13

 3.1.1 *Identifiers*..... 13

 3.1.2 *Definition of Transport Service*..... 14

 3.1.3 *Marshalling of Data Structures*..... 15

 3.1.4 *Handling of Inbound Traffic*..... 15

 3.1.5 *Authentication and Encryption*..... 16

3.2 HTTP MAPPING..... 16

 3.2.1 *Identifiers*..... 16

 3.2.2 *Definition of Transport Service*..... 16

 3.2.3 *Marshalling of Data Structures*..... 17

 3.2.4 *Handling of Inbound Traffic*..... 17

 3.2.5 *Authentication and Encryption*..... 17

3.3 SIP EVENTS MAPPING..... 17

 3.3.1 *Identifiers*..... 17

 3.3.2 *Definition of Transport Service*..... 18

 3.3.3 *Marshalling of Data Structures*..... 18

 3.3.4 *Handling of Inbound Traffic*..... 18

 3.3.5 *Authentication and Encryption*..... 18

3.4 WEB SERVICE EVENTING MAPPING..... 18

4. N-RSA REMOTE INTERFACE APPLICATION PROGRAMMER INTERFACES..... 19

5. CODE REPOSITORY COMMUNICATION PROTOCOL & API..... 20

5.1 COMMUNICATION PROTOCOL..... 20

5.2 MODULE API..... 20

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

6. REFERENCES..... 22

ANNEX A RETURN AND REASON CODES..... 22

Nokia Research Center / Helsinki
Dirk Trossen, Dana PavelFINAL
24.10.06

1. INTRODUCTION

This report presents the communication protocols used between the different components of the Nokia Remote Sensing Architecture (N-RSA).

The report will define a protocol framework for the event delivery within the N-RSA. In this framework, data structures and protocol operations will be described based on the event delivery paradigm that was introduced in [5]. In order to map the protocol framework onto different protocol bearers, we will also present a template for such mapping description. The main mappings will then be described based on the introduced mapping template. The introduced data structure for the event delivery framework will be used as a basis to describe the actual remote interfaces in N-RSA as instantiations of the defined event delivery data structure.

2. EVENT DELIVERY PROTOCOL FRAMEWORK

Based on the event delivery paradigm that was introduced in [5], this section will define a protocol framework for the event delivery in N-RSA. For that, we will first define the data structure for the protocol framework, before presenting the protocol operations. We will then define a mapping template to be used by the particular protocol bearers.

Note that many of the operations and structures described in this section are directly derived from major specifications of the SIP event framework [1][7][8]. The intention though is to define a more simpler event delivery framework that can be mapped onto different bearers (see Section 3). If mapped onto SIP events, the more advanced and dedicated event delivery features of the SIP event framework could be used in such environment (such as forking, migration of event servers, multiple state of subscriptions and so on).

2.1 Protocol Data Structures

As every protocol, the event delivery protocol, as outlined in [5], exchanges particular data structures between the originator and destination. These data structures for the particular methods (or messages), described in [5], will be defined in this section.

The APIs of remote interfaces of the N-RSA then will simply be instantiations of the particular data structures, without going into too much details as to how these instantiations will be conveyed to particular implementations (see Section 4).

The data structures in the following will be described using ANSI-C `typedef`, `struct`, `enum` and `union` constructs [6]. We will further use as data types `short int`, `long int`, `unsigned char` as well as the pointer to `unsigned char` to emulate octet strings. We assume 8 bit length for `unsigned char`, 16 bit length for `short int` and `enum`, and 32 bit for `long int`.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

With this in mind, the following data structures will be used in the event delivery protocol framework of N-RSA:

```
// This is a type definition for octet strings
// Note that the length is given upfront in contrast to C-like
// '\0' delimited strings!
// An implementation would insert the length first in the octet
// stream, followed by the appropriate number of octet obtained
// from the pointer.
typedef struct octet
{
    short int      length;          // length of the string
    unsigned char *string;         // must provide 'length' bytes
} OCTETSTRING;

// type of the method to be used in method struct
enum method_type
{
    method_SUBSCRIBE,
    method_NOTIFY,
    method_PUBLISH,
    method_CONFIRM,
    method_BYE
};

// type of the confirmation
enum confirm_type
{
    confirm_PUBLISH,
    confirm_OTHERS
};

// first the method-specific structures
typedef struct SUBSCRIBE
{
    short int      dialog_id;       // identifies dialog
    short int      CSeg;           // identifies transaction
                                        // within dialog, starting with
                                        // zero within each dialog
    long int       Expires;        // lifetime of the subscription
}SUBSCRIBE;

typedef struct NOTIFY
{
    short int      dialog_id;       // identifies dialog
    short int      CSeg;           // identifies transaction
                                        // within dialog, starting with
                                        // zero within each dialog
}NOTIFY;

typedef struct PUBLISH
{
```

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

```

        long int      e_tag;           // identifies publication
        long int      Expires;        // published state expiration
    }PUBLISH;

typedef struct CONFIRM
{
    confirm_type      type;           // type of confirmation
    union relation
    {
        struct dialog
        {
            short int  dialog_id;     // identifies dialog
            short int  CSeg;          // identifies transaction
                                        // within dialog, starting with
                                        // zero within each dialog
        };
        short int      e_tag;         // used for PUBLISH only
    }
    long int          Expires;        // actual expiration of request
    OCTETSTRING       ret_code;      // return code of confirmation
}CONFIRM;

typedef struct BYE
{
    short int         dialog_id;     // identifies dialog
    short int         CSeg;          // identifies transaction
                                        // within dialog, starting with
                                        // zero within each dialog
    OCTETSTRING       reason;        // reason code for termination
}BYE;

// now the structure for the entire method
struct Method
{
    method_type       type;           // type of method
    OCTETSTRING       FROM;          // originator
    OCTETSTRING       TO;            // receiver
    OCTETSTRING       event_name;    // name of the event
    union Method_specifics
    {
        SUBSCRIBE     sub;
        NOTIFY         not;
        PUBLISH        pub;
        CONFIRM        conf;
        BYE            bye;
    };
    OCTETSTRING       event_body;    // here's the event info
};

```

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

2.2 Endpoint Identifiers

In the above defined data structures, endpoint identifiers are merely defined as octet strings (`FROM` and `TO` field in the `Method` structure). However, different endpoint identifiers can exist depending on the particular transport bearer. N-RSA will define mappings onto TCP/IP, HTTP, and SIP events, which will result in the following possible endpoint identifiers:

- *IP addresses* for TCP/IP in the well-known format
- *URL:port* for HTTP, such as http://gateway_xyz@sensorix.com:9000
- *SIP URI* [7] for SIP events, such as `sip:gateway_xyz@sensorix.com`

It is assumed that the particular endpoint identifiers exist and that they are properly registered with the particular registration system (e.g., DNS or SIP location service). It is not within the scope of the specification how the assignment of the endpoint identifiers will happen since this concerns the actual deployment of the system rather than the design.

2.3 Protocol Operations

Apart from the actual data structures, the protocol operations for the event delivery protocol framework are important to be defined. Although already given informally in [5], the following subsections will define the protocol operations more precisely through message sequence charts.

2.3.1 Subscription & Notification

The message sequence chart of the subscription and notification operation can be seen in Figure 1.

The client (identified in the `FROM` field of the data structure) sends a well formed `SUBSCRIBE` method to the server (identified in the `TO` field of the data structure) to a particular event (identified in the `event_name` field of the data structure). The `event_name` field is given by the particular remote interface that is implemented over the event delivery protocol.

If the subscription is a new one, the `dialog_id` field is set to `0xffff`, otherwise the identifier of an existing dialog is inserted in this field. Within the particular dialog, the current transaction value is inserted in the `CSeq` field (starting with zero for each dialog). The `Expires` field contains the expiration time of the subscription (see also below).

Upon reception at the server, the `dialog_id` field is used to identify an existing dialog. If the value `0xffff` has been used, a new dialog is created at the server with a locally unique `dialog_id` value. The subscription information, i.e., `event_name` and `event_body`, is extracted and interpreted as required, depending on the particular remote interface that is implemented. Further, the server determines an expiration time for the subscription, which

must not be larger than the value given in the `Expires` field of the subscription but can be smaller.

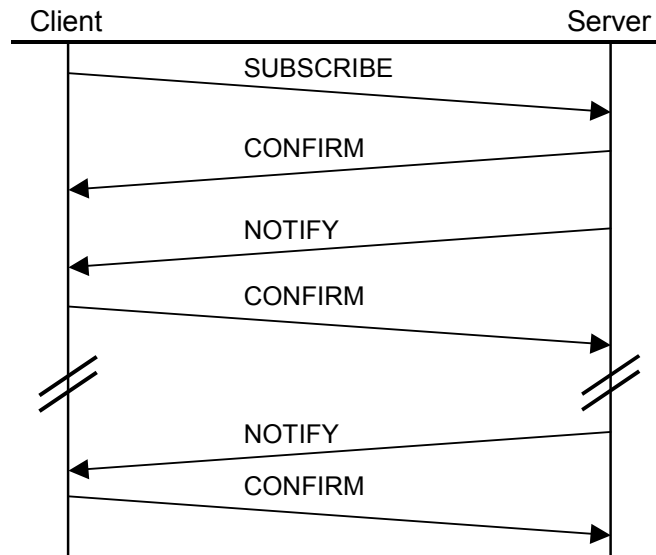


Figure 1 Protocol Operation for Subscription & Notification

If the subscription was successful, the server sends back a CONFIRM method, carrying the server’s address as `FROM` field and the client address as `TO` field, with reason code `200_OK` in the `ret_code` field. In all other cases, the appropriate reason code is inserted (see ANNEX A for a list of reason and return codes). The `dialog_id` field carries the identifier of the dialog and the `CSeq` field the transaction identifier, given in the SUBSCRIBE method. The `Expires` field indicates the actual expiration time of the subscription that was determined at the server.

After a positive confirmation, the server generates an initial NOTIFY method, indicating the current state of the subscription. For that, the `FROM` and `TO` fields carry the server’s and client’s address respectively. The `dialog_id` and `event_name` field of the particular subscription is used, while the `CSeq` value is derived from the previous value (incremental increase). The `event_body` entry is generated by the server according to the particular semantics of the remote interface that is implemented over the event delivery protocol.

The client responds to the NOTIFY method with a CONFIRM method similar to the first CONFIRM (`FROM` and `TO` field reversed and using the `dialog_id` and `CSeq` fields of the NOTIFY method).

If the `Expires` field of the SUBSCRIBE method indicated a value of zero, the subscription dialog is terminated after the initial NOTIFY method (one-shot subscription). Otherwise, the

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

`Expires` field is used to determine the time that the subscription is upheld by the server (provided that no other reason of failure occurs). As noted above, the actual lifetime of the subscription is eventually determined at the server side and returned to the subscriber in the `Expires` field of the CONFIRM method. During the lifetime of the subscription, additional NOTIFY methods can be generated at the server side, depending on the particular semantics of the subscription. In this case, the same parameters as in the initial NOTIFY will be used. Only the `CSeq` field is incremented appropriately with each NOTIFY method, indicating another transaction within the dialog. Also the `event_body` field will contain the particular event information for the particular notification. Again, the client responds to the NOTIFY method with a CONFIRM method similar to the first CONFIRM (`FROM` and `TO` field reversed).

2.3.2 Termination of Subscription

An existing subscription can be terminated either from the client or server side. Examples for the former are application reasons (e.g., no particular interest in the subscription). Examples for the latter could be resource reasons, temporary or permanent unavailability of resources to determine the subscription state or other server failures.

Figure 2 shows the message sequence chart for the client-side termination. In this case, the client (identified in the `FROM` field) sends a BYE method to the server (identified in the `TO` field). The `dialog_id` identifies the dialog to be terminated, while the `CSeq` field identifies the transaction appropriately (i.e., in the relation to former transactions). The `event_name` field is taken from the subscription to be terminated. The `reason_code` field contains codes according to ANNEX A, while the `event_body` field is used to further describe the reason of termination, if necessary.

Upon reception of the termination request, the server determines the subscription to be terminated via the `dialog_id` field. If found the subscription is terminated internally.

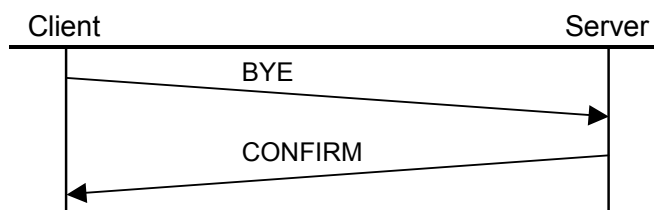


Figure 2 Protocol Operation for Client-side Termination of Subscription

In any case, the server responds with a CONFIRM method, using the `FROM`, `TO`, `dialog_id` and `CSeq` fields as given in the BYE method (of course, the `FROM` and `TO` fields are exchanged to reverse the direction). In case of an error code (i.e., different return value than `200_OK`), the server may add additional information in the `event_body` field. The `Expires` field is set to zero.

Upon reception of the CONFIRM method at the client, the appropriate subscription information is terminated at the client, too (in case of a successful confirmation). Otherwise, the subscription is terminated and the reason code is appropriately rendered to upper layers.

In case of a server-side termination (see Figure 3), the operation is similar to the client-side termination. The difference lies in the reason codes for the termination (expressed in the BYE method). Apart from this, the operation can be determined from the client-side termination above through reversing the roles of sender and receiver.

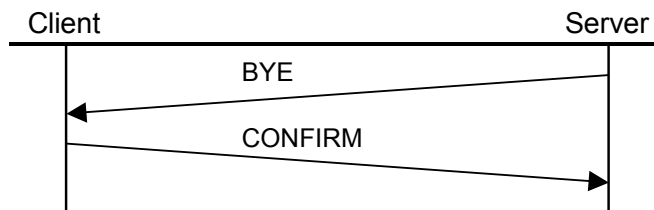


Figure 3 Protocol Operation for Server-side Termination of Subscription

2.3.3 Publication

Within the N-RSA, certain entities need to publish information regarding certain resources (e.g., sensors) to particular other entities. For that, the event delivery protocol framework provides the PUBLISH method and the corresponding operation as shown in Figure 4.

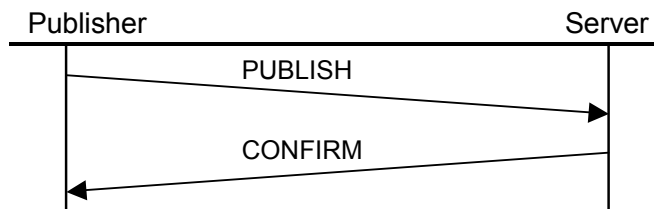


Figure 4 Protocol Operation for Subscription & Notification

In this, the publisher (identified in the FROM field) sends a PUBLISH method to the server (identified in the TO field). In case of an initial publication for the particular resource, the e_tag field is set to 0xffff. In case of a renewal or removal, the previous e_tag identifier is used (note that this mechanism serves a similar purpose than the dialog identifiers for the other methods).

The event_name field identifies the event to which the published information relates, while the event_body contains the actual published information. The Expires field indicates the expiration time of the publication. If the expiration time is set to zero, the published state is removed at the server. Otherwise, the published state is refreshed or modified. In the former

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

case, the event body is left empty (i.e., simply the expiration time of the already existing state at the server is refreshed). In the latter case, the new event information, contained in the `event_body` field, is used at the server.

After receiving the PUBLISH method, the server extracts the information and uses it appropriately in the context of the given event. The server uses the include `e_tag` field to relate the received publication to an existing one. If it indicates an initial publication (field set to `0xffff`), the server generates an `e_tag` field to be delivered in the response. Otherwise, the enclosed `e_tag` field in the publication is used to refresh, modify or remove the corresponding publication. Eventually, the server confirms the publication by sending a CONFIRM method back to the publisher. The method contains the `e_tag` field, generated at the server. This field is stored at the publisher for future use (e.g., removal or refresh of publications). The method further contains the actual expiration time, determined at the server, in the `Expires` field of the CONFIRM method. The value of this field must not be larger than the given `Expires` field value in the publication, but can be shorter.

2.4 Template for Event Delivery Mapping

The event delivery protocol framework, as described in the previous subsections, is meant to define a generic event delivery than can easily be mapped onto different transport bearers. In order to give a certain framework also for this part, i.e., the actual mapping, we will outline a template for such mapping within this subsection.

2.4.1 Identifiers

Each mapping has to define a proper endpoint identifier format. This format must be based on existing standards for endpoint identifiers. Typical examples for such endpoint identifiers are telephone numbers, URIs, or IP addresses.

Note that it is not within the scope of the current N-RSA as to how an application could map endpoint identifiers onto particular gateways. One could envision an application service that would allow for resolving particular locations (or proximities) into possible gateways (and their respectively supported endpoint identifiers). Further, a mapping between certain endpoint identifiers is not within the scope of the current N-RSA. However, such mapping and resolving service could easily be integrated as an application layer (from the N-RSA perspective) service.

2.4.2 Definition of Transport Service

The underlying transport bearer is assumed to provide an unreliable datagram transport service as a minimum. Hence, each mapping has to define such datagram transport service in the mapping specification. This includes the specific datagram format as well as the mapping of the data structures defined in Section 2.1 onto the defined datagram format. Further, any necessary transport information, such as port information, has to be defined in the mapping.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

Although reliable transport is desired, it is not mandatory. However, if the particular transport supports reliable transport, it should be used and therefore described in the mapping specification.

2.4.3 *Marshalling of Data Structures*

Since the ordering of data structures depends on the particular end system details (such as processor, its operation system, particular compiler), it is required to send any data structures in the N-RSA in a defined manner as a *marshaled transport data structure*, i.e., the data requires marshalling based on a particular transport syntax before being sent. Each transport mapping must specify what the basis for such marshalling is.

2.4.4 *Handling of Inbound Traffic*

Within the N-RSA, the gateways are mainly considered to be devices connected to a cellular operator network. A common problem in such deployments is the existence of an operator firewall that blocks inbound connection requests. Each mapping has to specify how this problem is solved with the particular mapping. Such solution must enable event delivery from and to the gateway within the N-RSA.

2.4.5 *Authentication and Encryption*

Authentication of the endpoints is crucial for remote sensing due to the potentially sensitive character of the transferred sensor information. Further, the actual data shall be transferred in an encrypted manner, if so desired by the participating endpoints. Hence, each mapping must define endpoint authentication and transport encryption methods.

3. EVENT DELIVERY MAPPINGS

This section describes the specific mappings of the event delivery protocol framework of Section 2 onto particular transport bearers. For that, we will use the mapping template of Section 2.4.

3.1 *TCP Mapping*

The Transmission Control Protocol (TCP) provides a stream-oriented transport protocol with reliability and congestion control mechanisms. It is therefore well suited to provide a simple transport bearer for event delivery. In the following, the necessary mapping of the event delivery protocol framework is presented.

3.1.1 *Identifiers*

The endpoint identifiers of this mapping are either DNS-conform hostnames, such as *gateway123.mysensor.com*, in order to abstract from the actually used IP address or direct IP addresses, where possible.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

The translation to the particularly used IP address is either done using standard DNS mechanisms or it is done within the transport implementation. The former is the case when having network entities with proper DNS registration and globally routable IP address (such as the application server, see also Section 3.1.4). The latter is done for cases in which the particular N-RSA entity does not have a globally unique routable IP address (i.e., DNS would not be available).

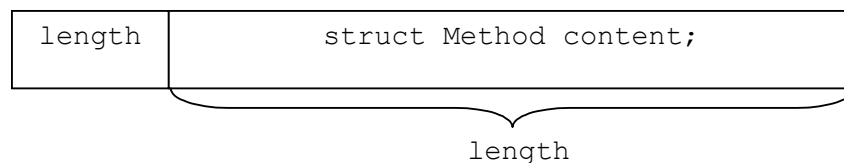
The case of non-globally routable IP addresses usually applies for the gateways due to the dynamic assignment of IP addresses by the mobile operator. In this case, the transport mapping implementation has to map the provided DNS-conform name onto the dynamically assigned IP address of the gateway. The assignment of DNS name to particular gateway connection (i.e., dynamic IP address) within the transport mapping implementation happens during the registration of availability, using the R_A interface [5]. Hence, when registering the gateway's resources with the application server, the event delivery mapping implementation caches the mapping of the particular transport connection (i.e., the dynamic IP address) onto the (DNS-conform) FROM field of the registration message. In the future, this assignment will be used for any communication to this gateway.

IP addresses (either IPv4 or IPv6, depending on the used network deployment) can be used for entities that have globally routable IP addresses (such as the application server, see also Section 3.1.4). When addressing the application server from the gateway's side, IP addresses should be used in order to avoid the usage of DNS at the gateway. In such cases, the IP address of the application server could be pre-configured at the gateway. Such pre-configuration is beyond the scope of this specification.

In conclusion, (globally routable) IP addresses and DNS-conform hostnames can be used for addressing the application server and the repository server, while the gateway is addressed through DNS-conform hostnames only (due to the dynamic IP address of the gateway in most mobile deployments). Given DNS-conform hostnames are resolved in corresponding IP addresses within the mapping implementation either using an internal resolving mechanism for the gateway's hostname or standard DNS.

3.1.2 Definition of Transport Service

Since TCP already provides a fully functional transport service, the only required mechanism is an appropriate message framing of the event delivery protocol framework data structure (as defined in Section 2.1). This is done through prefixing the actual data structure with a `length` field of 16 bit size as shown in Figure 5.



Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

Figure 5 Datagram Structure for TCP Mapping Case

In order to uniquely identify the local endpoint, it is required to specify a port in the establishment of the particular TCP connection. This mapping will use the port number 9000 for this purpose.

3.1.3 *Marshalling of Data Structures*

In this mapping, the marshaled data structures are constructed through readout of the defined data structures in Section 2.1 together with the particular transport data structure (see above).

The assumed byte ordering for the resulting marshaled transport data structure is little endian.

3.1.4 *Handling of Inbound Traffic*

In order to establish a TCP connection between two entities in the Internet, at least one entity needs to have a globally routable IP address together with a reachable port (i.e., no firewall blocking inbetween). Since this isn't the case for the gateways due to the dynamic assignment of the gateway's IP address by the mobile operator, it is imperative that the application server's as well as the code repository's IP address are such globally routable ones.

It is not within the scope of this specification as to how the gateway obtains knowledge of the IP address information. Possible ways are static configuration (e.g., through a mobile phone user interface) or via SMS, i.e., the application server may wake up the gateway with an SMS that includes the IP address of the server.

The first communication between gateway and application server is the identification of the gateway towards the application server (see section below). After successfully identifying and authorizing the connection from the gateway, the next communication is the registration of gateway resources with the application server (using the R_A interface [5]). This publication of resources will happen from the gateway towards the (globally reachable) application server.

For that, the gateway will establish an initial TCP connection to the application server, with the FROM field of the publication operation (see Section 2.3.3) being the gateway-specific DNS-conform hostname. The TO field carries either the application server's IP address or the DNS-conform identifier. After accepting the incoming connection request at the application server, the incoming connection and the provided DNS-conform hostname are cached internally at the application server. This cached connection identifier (also called *connection context information*, in socket-compliant implementations usually being a `socket` identifier) is used for any future communication with the gateway. Hence, when an event operation (e.g., subscription) is issued at the application server, the gateway's DNS-conform hostname is mapped onto the stored connection-specific identifier.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

As for the case of communicating between code repository server and gateway, such initial TCP connection is established in the process of the first subscription/notification operation on the C_R interface.

In the case of a TCP connection failure throughout the lifetime of the TCP connection, the gateway must re-establish another TCP connection. In case of gateway<->application server communication, this will be done through re-performing the registration procedure (i.e., the TCP connection is again established as above). In the case of gateway<->code repository communication, such initial connection is automatically restored with the next code download.

In such case of connection failure, the connection context information (see above) at the application server or repository server is removed.

3.1.5 Authentication and Encryption

The first communication between gateway and application server is the identification of the gateway towards the application server (see section below). For this, the IMEI information is used, i.e., the gateway sends its own IMEI to the application server in a plain 15 bytes information field. It is assumed that the application server holds some information (in a text file) about the registered gateways and their IMEI information. Upon reception and positive verification of the IMEI, the application server proceeds with the publication procedure. Otherwise, the connection is terminated. In the future, other forms of authorization are envisioned.

Payload encryption is realized through standard IPSEC mechanisms.

3.2 HTTP Mapping

The Hyper-text Transfer Protocol (HTTP) provides an end-to-end push transport mechanism based on underlying TCP connections. Hence, similar problems apply to this mapping with respect to identifiers and inbound traffic.

3.2.1 Identifiers

The identifiers are, similar to the TCP mapping, DNS-conform hostnames. Due the stacking of HTTP over TCP, the same discussions as in Section 3.1.1 apply with respect to globally routable IP addresses.

3.2.2 Definition of Transport Service

HTTP provides a fully functional transport service that is used in this mapping. Specifically, we use the PUT method of HTTP, where the body of the method contains an XML formatted version of the N-RSA data structures described in Section 2.1. For that, an appropriate XML schema is applied that uses the data structure type names in Section 2.1 as XML element names with the appropriate values carried in these elements.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

3.2.3 *Marshalling of Data Structures*

The message bodies, obtained through abovementioned XML schema are fed into available HTTP protocol stacks, which will apply the appropriate marshalling of the given data structures.

3.2.4 *Handling of Inbound Traffic*

The same problems apply for HTTP as in the TCP case, i.e., HTTP endpoints are not reachable in cases where these endpoints do not have globally routable IP addresses.

Due to the TCP connection re-usage in HTTP/1.1, an inbound solution already partially exists. In other words, the first HTTP PUT method that is used in the initial registration of the gateway would establish an outbound TCP connection to the application server. This TCP connection could then be re-used for further HTTP operations. However, this would assume proper settings for the keep-alive timer in the HTTP protocol stack.

For inbound connections however, such solution does not really help without modifications to the HTTP stack. Since the initial HTTP operation was outbound (from a gateway's point of view), the application server would initiate a separate TCP connection for its own outbound (from the application server's perspective) operation. Hence, without a globally routable IP address for the gateway, such operation would fail. In order to tackle this problem, a modification to the HTTP protocol stack is required, re-using the TCP connection of incoming HTTP requests for outgoing HTTP requests as well. It is possible that such optimization already exists in available HTTP protocol stacks.

3.2.5 *Authentication and Encryption*

Endpoint authentication and payload encryption is realized through standard HTTP security mechanisms.

3.3 *SIP Events Mapping*

SIP Events (defined in [1]) provide an Internet-wide event delivery framework that is well suited for the purposes in N-RSA. Since the N-RSA event delivery framework borrowed the main concepts from SIP events, the mapping onto SIP events is rather straightforward.

3.3.1 *Identifiers*

The endpoint identifiers of this mapping are SIP URIs, compliant to [7], such as *sip:gateway123@mysensor.com*. It is necessary at boot-up of a particular gateway that the gateway registers its AOR (address of record) and contact address with the SIP proxy of the particular domain. It is not within the scope of the N-RSA, where and who is providing such SIP proxy. However, a typical, SIP-based, deployment of the N-RSA would most likely include a SIP proxy, although such SIP proxy can also be used for other purposes, i.e., SIP-based services, within the given domain.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

However, one has to keep in mind that the provided IP contact address needs to be reachable for the SIP proxy, which might require the SIP proxy to be deployed by the cellular operator. In this case, an appropriate agreement might be necessary as to use the operator's SIP infrastructure to perform the N-RSA operations. If the gateway's IP address is globally routable, a non-operator SIP proxy might be used.

3.3.2 Definition of Transport Service

The SIP event framework, as defined in [1], provides a transport-level service for delivering the necessary methods of the N-RSA event delivery framework to the components in the N-RSA. For that, the methods of the N-RSA event delivery framework are simply mapped on the methods described in the SIP event framework and its related standards (such as [8]). It is a choice of implementation as to which particular transport protocol, e.g., UDP, TCP or SCTP, is used.

3.3.3 Marshalling of Data Structures

The marshalling of the N-RSA data structure is simply performed through mapping the data fields onto appropriate SIP message headers and bodies, according to [1][7][8]. Due to the chosen names in the N-RSA data structures, such mapping is rather straightforward to perform. The then derived SIP message structures are fed into the chosen SIP protocol stack.

3.3.4 Handling of Inbound Traffic

Due to the registration of the gateway with the particular SIP proxy of its domain, the event delivery communication can be established between the application server and the gateway, assuming a service level agreement between the application server and the SIP proxy. In the case of a private IP contact address, it is assumed (as discussed in Section 3.3.1) that the SIP proxy is deployed by the cellular operator, which ensures the proper delivery even in this case.

3.3.5 Authentication and Encryption

Endpoint authentication and payload encryption is realized through standard SIP mechanisms [7].

3.4 Web Service Eventing Mapping

WS Eventing [9] as a transport is currently not defined. It would allow for defining an event mapping onto a Web Service compliant bearer. Compared to the HTTP mapping (see Section 3.2), the WS Eventing would wrap the messages into appropriate SOAP envelopes. However, similar problems with respect to globally routable IP addresses would apply.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

4. N-RSA REMOTE INTERFACE APPLICATION PROGRAMMER INTERFACES

The APIs for the remote (and the local) interfaces within the N-RSA can easily be constructed through instantiating appropriate data structures that are fed into the appropriate components of the N-RSA [5]. These instantiations are based on the data structures defined in Section 2.1 and the given interface specifications in [5].

The protocol operations can be implemented through multithreaded objects using callback methods for the notification parts. The callback methods could carry the relevant data structure fields as parameters. The interface descriptions in [5] clearly define the relevant parameters for each N-RSA interface that has to be set by the particular API method (such as event name of the data structure).

However, the actual signatures of such methods that fill these data structures and implement the callbacks are not within the scope of the report since they concern aspects of the implementation architecture rather than the system architecture.

Nokia Research Center / Helsinki
Dirk Trossen, Dana PavelFINAL
24.10.06

5. CODE REPOSITORY COMMUNICATION PROTOCOL & API

Two pieces need consideration with respect to the code repository, namely the communication protocol for downloading code modules between the gateway and the repository server, and the API defined for the code modules in order to properly invoke the code. Note that the pieces of registering code modules with the application server are covered by the same R_A interface that is used for registering sensor information (i.e., the same payload format is used).

5.1 Communication Protocol

The communication protocol between the code repository components in the gateway and the code repository server is implemented through the C_R interface, as described in [5].

Hence, the N-RSA event delivery framework is used for the communication between the N-RSA components, i.e., the gateway and the repository server, with the parameters given in the C_R interface specification [5]. In this, the SUBSCRIBE body carries the particular label whose aggregation functionality needs to be downloaded (see also [10] for how to derive this label as part of the query resolving process). The label is denoted in an appropriate format, using an XML notation. The exact definition of this XML format is a matter of implementation. The NOTIFY response of the interface carries in its body the aggregation code in JAR format.

5.2 Module API

As described in Annex A of [5], there are two possibilities of realizing the N-RSA in the gateway, namely using Java Imlets for the entire middleware or using a C-based AM (Application Module) with Java code module download. In the first case, it is described in [5] that a code module download would not be feasible due to the restriction in the N12 of having only one JAR file active at the same time. Hence, we assume the second case, i.e., having a C-based middleware implemented as an application module.

In such case, the aggregation code modules are assumed to be JAR files, i.e., packaged Java Imlets, that are downloaded and executed by the AM. For the download, the communication protocol described in the previous section is used. After receiving the code in the body of the NOTIFY message, the code repository component [5] of the middleware needs to instantiate and execute the code module with the internal Java VM.

For communicating with the Java code, the usage of JNI (Java Native Interface) is assumed. The Java aggregation code will take as input an array of the input context information and return a single piece of context information. Input and output information are realized as a structure that carries as a first item a pointer to the context label (in order to associate the information to a type in the code) as well as a pointer to the actual context data (which could be a single item, an array or another struct). The format of the actual context data is determined by the aggregation code by the given context label.

Nokia Research Center / Helsinki
Dirk Trossen, Dana PavelFINAL
24.10.06

With this, the following signature (in pseudo-code) is defined for the code module method

```
struct context
{
    char *label;
    void *context_info;
};

struct context *JNI_code_module(int no_context, struct context
*context_pieces)
```

with `no_context` being the number of input context pieces.

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

6. REFERENCES

- [1] A. Roach, "SIP-Specific Event Notification", Internet Society, RFC 3265, July 2002
- [2] J. Hyryläinen, I.Jantunen "SSI Protocol Specification V1.0", April 2005, available at http://ssi-protocol.net/SSI%20protocol%20specification_10a-2.pdf
- [3] D. Trossen, D. Pavel, "N-RSA High-Level System Architecture", N-RSA 2004 project report, January 2004
- [4] Aplicom, "Aplicom M2M System Protocol 2 Socket Interface User Manual", available at http://www.aplicom.com/gsm_modules_documentation.html, 2006
- [5] D. Trossen, D. Pavel, "N-RSA High-Level System Architecture: Functionality & Interface Description", N-RSA project report, February 2004
- [6] B. Strausstrup, "The C++ programming language", Addison-Wesley, 1998
- [7] J. Rosenberg et al. "SIP: Session Initiation Protocol", Internet Society, RFC 3265, July 2002
- [8] A. Niemi, "An Event State Publication Extension to the Session Initiation Protocol (SIP)", Internet Draft, Internet Engineering Task Force, Work in Progress, February 2004
- [9] L. F. Cabrera, "Web Services Eventing (WS-Eventing)", available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/WS-Eventing.asp>
- [10] D. Pavel, D. Trossen, "Abstraction Model in the N-RSA System Architecture", N-RSA 2004 project report, May 2004

ANNEX A RETURN AND REASON CODES

The reason codes for the CONFIRM and BYE methods are mainly aligned to the specification defined in RFC 3261 [7] and RFC 3265 [1]. The following list gives an overview of the main used reason codes and their meanings.

"202 Accepted"

Reason code has the same meaning as that defined in HTTP/1.1

"489 Bad Event"

Used to indicate that the server did not understand the event package specified in the `Method.event_name` field.

"200 OK"

Indicates success of the operation.

"400 Bad Request"

Indicates malformed request, i.e., server could not interpret request.

"401 Unauthorized"

Originator is not authorized to issue the particular request.

"404 Not Found"

Nokia Research Center / Helsinki
Dirk Trossen, Dana Pavel

FINAL
24.10.06

One or more of the resources, required for the request, are not found. The body of the CONFIRM operation should indicate the particular resource(s).

“408 Request Timeout”

The method request timed out. The body of the CONFIRM should include further explanation of the timeout.

“413 Request Entity Too Large”

Indicates that the body of the method is too long for the server to properly process.

“416 Unsupported URI Scheme”

The provided identifier (FROM and/or TO) is not supported by the server.

“487 Request Terminated”

Given as reason code in BYE transaction. The body may contain further information on the termination reasons, which could be rendered to the user