Aalto University
School of Electrical Engineering

Department of Communications and Networking
S-38.3159 Protocol Design P
Spring 2014

Sensor Fusion and Optimization
Assignment 1: Protocol specification

Student 1:	Riku Lääkkölä	69896S
	riku.laakkola@aalto.fi	
Student 2:	Tero Marttila	78949E
	tero.marttila@aalto.fi	
Student 3:	Tero Paloheimo	78510C
	tero.paloheimo@aalto.fi	

1 Introduction

This document describes a UDP-based protocol for use in a client-server based publish-subscribe model. The server receives sensor updates, and publishes the sensor values to subscribed clients. Individual clients may query for the list of available sensors, and update their set of subscribed sensors. The server will monitor sensor availability, and update the client when sensors become available or unavailable. The protocol is designed for real-time purposes, and as such, provides ordered unreliable transmission, periodically transmitting the newest state.

The bidirectional control streams are identified by a type code and separate sequence counters for client and server transmissions, which are acknowledged by the recipient. The subscription protocol is based on the use of soft state, with the client and server transmitting their full subscription state on each change. It is expected that the amount of subscription-related traffic will remain minor compared to the actual **publish**-traffic.

The unidirectional data stream is used by the server to publish the most recent sensor values to the client. The client will periodically acknowledge the received sensor publishes, which is used by the server as a feedback channel for keepalive and congestion control purposes. It is vitally important for the server to handle clients that disappear, timing out their subscriptions.

The messages are transmitted as UDP packets between the client and server, using a fixed port on the server, and an arbitrary port on the client. All messages are transmitted and received using the same pair of addresses/ports. If the client changes to a different address, it must create a new subscription to the server, and the previous one will time out.

2 Message format

The protocol is based on a hybrid binary/text encoding, using a fixed-size binary header for transport control purposes, and a JSON-encoded variable-length payload.

The message header format is described in Figure 1. The Magic-field is an identifier for our protocol and also works as a version number. The Type-field denotes the message type (subscribe, unsubscribe, etc.) and the A-field denotes the *ackthis* bit. The remaining bits are reserved for future use. The following fields are for sequence numbers: **Ack-Seq** is used for acknowledging receiver sequence numbers, and **Seq** is used for the sender sequence number.

The range of possible messages and associated header field values are listed in Table 1.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1|     Magic     |      Type     |A|          Reserved           |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
2|                            Ack-Seq                            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3|                              Seq                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4|                            Payload                            |
 |                              ...                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Message format

Table 1: Message types. The header field values are shown in Table 2.

		Header field value				
Message	Direction	T	A	Ack-seq	Seq	Payload
subscribe-query	C ⇀ S	0	0	0	0	
subscribe-queryresponse	C ↽ S	0	0	0	0	x
subscribe-request	C ⇀ S	0	0	0	*C-sseq*++	x
subscribe-response	C ↽ S	0	0	*C-sseq*	*S-sseq*++	x
subscribe-update	C ↼ S	0	0	0	*S-sseq*++	x
subscribe-ack	C ⇁ S	0	0	*S-sseq*	0	
publish	C ↼ S	1	0	0	*S-pseq*++	x
publish (/w ackthis)	C ↼ S	1	1	0	*S-pseq*++	x
publish-ack	C ⇁ S	1	0	*S-pseq*	*nrecvd*	
teardown	C ⇀ S	2	0	0	0	
teardown-ack	C ↽ S	2	0	0	0	

Table 2: The header field values.

T Type

A The "Ack this" flag field (used for **publish** only)

C Client

S Server

sseq subscribe message specific sequence number

pseq publish message specific sequence number

nrecvd Number of publishes received by the client so far

Below is a list of the various message variants and their semantics:

subscribe-query Client initiated query of the available sensors. The server sends the sensor list and the client processes it.

subscribe-queryresponse Response from the server to *subscribe-query*, contains a list of available sensor names. The payload must have the following format: `["sensor1", "sensor2",...]`.

subscribe-request A message sent by the client to update its sensor subscription. Contains a list of the sensors to which the client wants to subscribe. This message must be acknowledged by the server. If the client wants to subscribe to one or more sensors, the payload must be a list with the sensor names: `["sensor1","sensor2",...]`. If the client wants to subscribe to all sensors, the payload must be `true`. Unsubscribing, that is, subscribing to no sensors is done by sending an empty list. The payload must thus be `[]`.

subscribe-response Server acknowledgement to the *subsribe-request* message. The message contains a dictionary which contains all the sensors and separately specifies the subscribed sensors. A subscribed sensor has the dictionary value of `true` and an unsubcribed has the value `false`. Thus the payload must have the following format:
`{"sensor1":false,"sensor2":false,"sensor3":true,...}`.

subscribe-update A message from the server containing an updated list of sensors. This message is sent whenever the sensors are updated, that is, a sensor is added or removed. This message needs to be acknowledged by the client and is sent until an acknowledgement is received. The payload must have the same format as the *subscribe-response* message.

subscribe-ack Client initiated message which acknowledges the *subscribe-update* message.

publish The server-initiated message contains the values of the subscribed sensors. If this message is lost, it is not retransmitted but new values are sent instead. The payload is a list with the sensor values (dictionary) from each sensor, an example: `[{...},{...}]`.

publish (/w ackthis) Like the "normal" *publish* message but this must be acknowledged by the client. The acknowledgement is used by the server to check whether the other party is still available, and measure congestion. This message has the same payload format as the unacknowledged *publish* message.

publish-ack Message from the client which acknowledges the previous *publish (/w ackthis)* message.

teardown A message sent by the client indicating that is wants to end the communication with the server. This message must be acknowledged by the server. A client sends this message until it receives the *teardown-ack* message as an acknowledgement. If the server receives this message from an unknown client, it will respond with a *teardown-ack* message.

teardown-ack Server response to the *teardown* message sent by a client. After sending this message, the server timeouts for T_t seconds, and if no retransmits of *teardown* arrive, the connection is terminated and resources are freed.

The *subscribe-** messages have a timer mechanism. When a message is sent with an incremented sequence number, the timer is set. Receiving a message with the matching *type* and *ack-seq* field will clear the timer. If the timer expires after T seconds, the previously sent message will be retransmitted (using the same seq). Handling of received messages is based on the *seq* field. The highest received *seq* is recorded, acked and processed. A duplicate *seq* corresponding to the updated seq is considered a duplicate, and is acked without being processed. A lower *seq* than recorded indicates packet reordering and is dropped.

3 Protocol description

3.1 Subscribing and unsubscribing

When a client wants to subscribe to some sensors it will start by sending a *subscribe-query* message to the server. The server will then reply with a *subscribe-queryresponse* message, which contains the list of currently available sensors. The client subscribes and unsubscribes to sensors by sending a *subscribe-request* message, which contains a list of sensors to which the client wants to subscribe. The server will then respond with the *subscribe-response* which contains the sensors to which the client is subscribed. Finally, the client responds to the previous message with the *subscribe-ack* message to acknowledge its reception of the message. A normal sequence of message sent when a client subscribes to some sensors is shown in Figure 2. The same sequence with lost packets is shown in Figure 3.

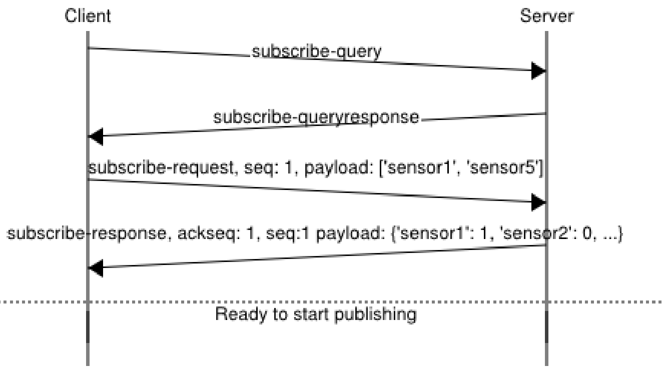
Client
Server
subscribe-query
subscribe-queryresponse
subscribe-request, seq: 1, payload: ['sensor1', 'sensor5']
subscribe-response, ackseq: 1, seq:1 payload: {'sensor1': 1, 'sensor2': 0, ...}
Ready to start publishing

Figure 2: A normal subscription sequence.

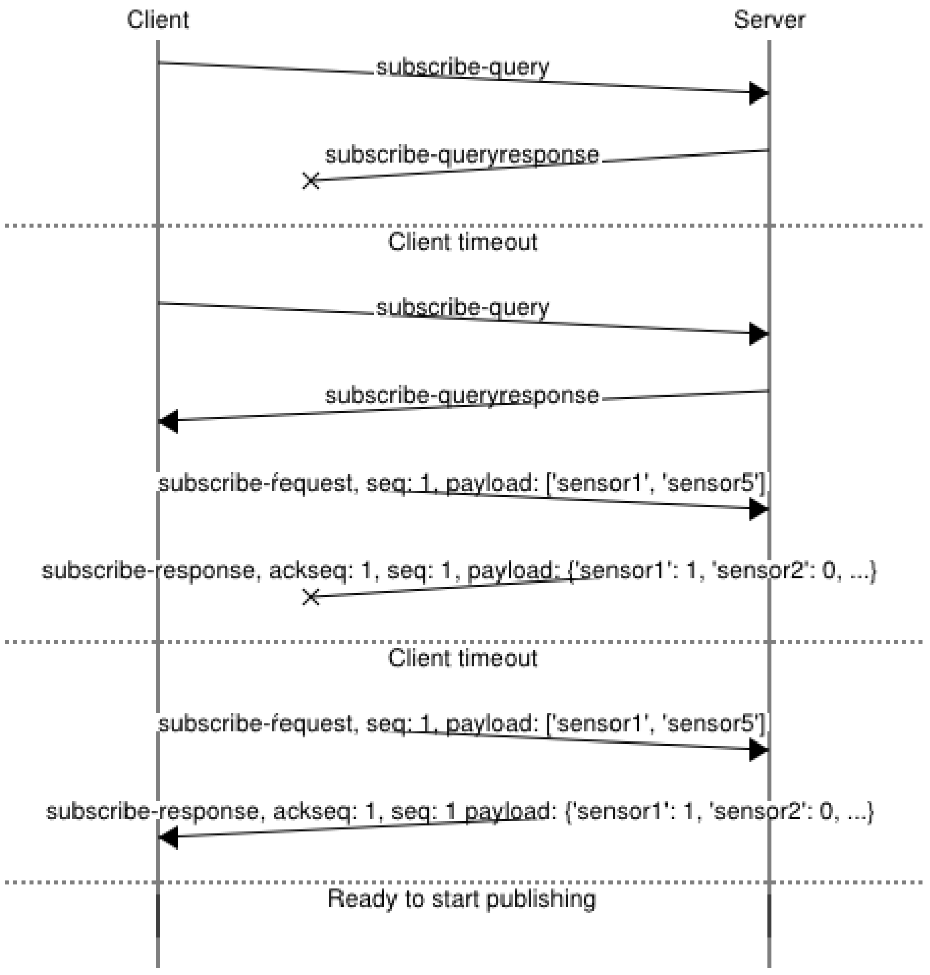
Client
Server
subscribe-query
subscribe-queryresponse
Client timeout
subscribe-query
subscribe-queryresponse
subscribe-request, seq: 1, payload: ['sensor1', 'sensor5']
subscribe-response, ackseq: 1, seq: 1, payload: {'sensor1': 1, 'sensor2': 0, ...}
Client timeout
subscribe-request, seq: 1, payload: ['sensor1', 'sensor5']
subscribe-response, ackseq: 1, seq: 1 payload: {'sensor1': 1, 'sensor2': 0, ...}
Ready to start publishing

Figure 3: A subscription sequence with packet loss.

When the available sensors change, the server will send a new list of sensors to the clients with the *subscribe-update* message. The client will acknowledge the reception by sending the *subscribe-ack* message. If the client now wants to change its subscription, it may send a new *subscribe-request* message with a new sensor list, which may contain any arbitrary subset or superset of the previous subscription sensor set, including an empty set to unsubscribe from all sensors.

A client terminates the connection by sending the *teardown* message to the server. It expects a response from the server and continues sending the same message until a response is received. The server responds by sending a *teardown-ack* message and waits for timeout T_t for retransmits of the *teardown* message. If no retransmits arrive, the server may release its resources related to the client. If the server receives a *teardown* from a client that has no resources currently allocated for it, the server only responds with the *teardown-ack*

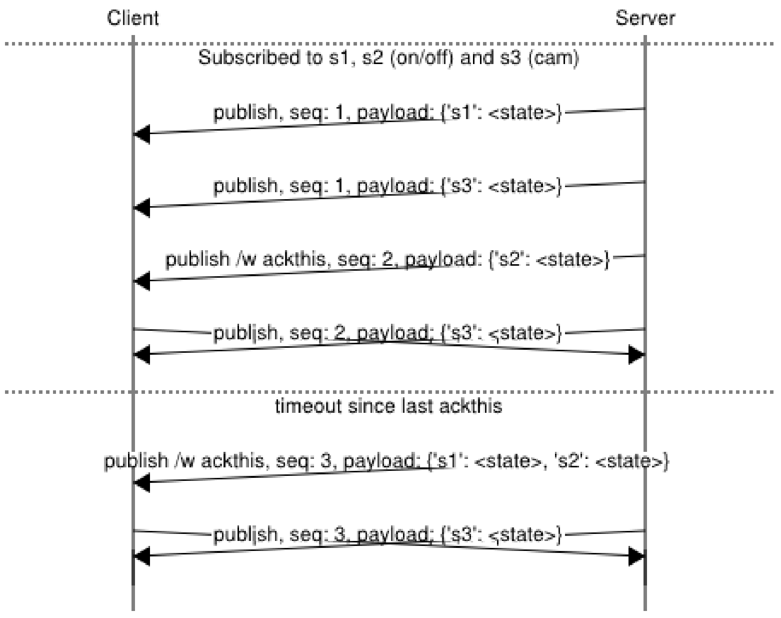
Client
Server
Subscribed to s1, s2 (on/off) and s3 (cam)
publish, seq: 1, payload: {'s1': <state>}
publish, seq: 1, payload: {'s3': <state>}
publish /w ackthis, seq: 2, payload: {'s2': <state>}
publish, seq: 2, payload: {'s3': <state>}
timeout since last ackthis
publish /w ackthis, seq: 3, payload: {'s1': <state>, 's2': <state>}
publish, seq: 3, payload: {'s3': <state>}

Figure 4: A publish message sequence.

3.2 Publish

Whenever a new sensor value is received at the server it will be sent to the clients who are subscribed to the selected sensor. The updates are sent with the *publish* message, which contains the current value of the sensor. These messages are not retransmitted and generally not acknowledged, except in the case of the on/off -type). Publish messages with the "ackthis" bit are sent every T_s seconds even if there is no sensor data to publish. The payload of these messages are last states of the subscribed on/off sensors. These messages must be acknowledged by the client with the *publish-ack* message, but the server may continue sending normal *publish* messages without waiting for the ack to arrive. A publish message sequence is shown in Figure 4.

The purpose of these messages is to act as a "keepalive" mechanism (and also double as a reliability mechanism for the on/off sensors). If acknowledgements from a client are not received within a time of $N \cdot T_s$ seconds for some value of N, the server will timeout the client. The same mechanism also works in the opposite direction, if publish messages are not received from the server within $M \cdot T_s$ seconds for some value of M, the server is considered to be unavailable and the client will time out. The time it takes for an acknowledgement to be received after the sending of a publish message is used to estimate the RTT. The *publish-ack* message sent by the client includes both the number of messages sent by the server (*ack-seq*) as well as the number of publish messages received by the client (*seq*). The server may use this information to determine packet loss for congestion control purposes.