Verification of QoS 2 level Scenarios

Irem Ozen iremozen123@gmail.com

***Abstract*— Message Queue Telemetry Transport (MQTT) is a popular security protocol that belongs to the internet of things (IoT), which we use frequently today. Nowadays, the spread of IoT usage has increased the requirement for cybersecurity. This study aims to determine verification of Quality of services 2 (QoS2) level protocol which is a security protocol of MQTT. To carry out this study, three different scenarios were created about Quality of Service 2 level which is an important main security protocol for this document. These scenarios were examined one by one and attacks were analysed. While possible attacks were executing, claim events were used which are frequently important security control events. All of these processes were analyzed on a tool called Scyther. Consequently, there are differences both theoretically and practically, but there is still no certainty in terms of safety. But this has been achieved for the purpose of the study. The scenarios have been evaluated and possible attacks have been interpreted. This study has therefore given us an idea of the system's functioning.**

***Keywords— IoT, MQTT, QoS 2 level, Scyther Tool***

# Introduction

IoT(internet of things) is one of the most important systems that has entered our lives and has become mainstream at a rapid rate. IoT represents devices that builds a smart network that share information. These devices communicate with each other through various communication protocols. Nowadays, every electronic device is becoming intelligent, that’s why these devices are now able to communicate with each other. Many technologies and ideas on IoT are being developed and some are being used.

The most important communication protocol is MQTT for IoT technology. MQTT, the abbreviation for the concept of "Message Queuing Telemetry Transport", is a telemetry message protocol based on the principle of broadcasting and subscribing. This protocol provides communication between machines. Therefore, it has a very important role in IoT.

# Message Queue Telemetry Transport

The MQTT protocol [4] is described as a lightweight broker-based publish/subscribe messaging protocol that was designed to allow devices with small processing power and storage, such as the devices IoT is composed of, to communicate over low-bandwidth and unreliable networks. MQTT is nowadays used in many business scenarios, for example, Facebook Messenger.

The publish/subscribe message pattern [4], on which MQTT is based provides for one-to-many message distribution with three varieties of delivery semantics, based on the level of Quality of Service(QoS) expected. The protocol additionally defines the message structure needed in communications between client, that is end-devices responsible for generating data from their domain and servers, which are the system components responsible for collating source data from clients/end-devices and distributing these data to interested subscribers.

## Quality of Services (QoS)

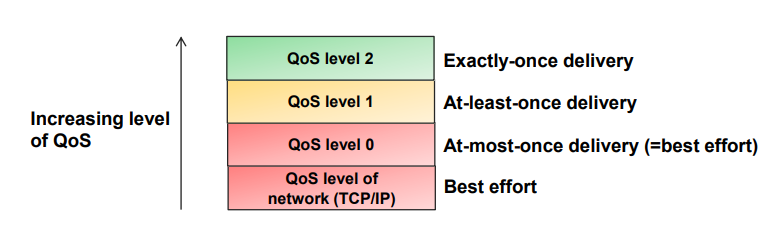
QoS is a network service which aims to reduce time loss by prioritizing applications on a network. It corresponds to various techniques that prioritize a traffic or program running over a network connection. There are three different level of QoS. These levels determine how the MQTT protocol manages content. High-level QoS is more reliable, but may experience more delay and needs more bandwidth. That’s why subscribers can define their own QoS levels according to what they want to receive.

QoS 0 (At most once) - Messages are delivered according to the delivery guarantees of the underlying network (TCP/IP). Message loss can occur.

QoS 1 (At least once) - Messages are guaranteed to arrive, but there may be duplicates.

QoS 2 (Exactly once) - This is the highest level that also incurs most overhead in terms of control messages and the need for locally storing the messages. Exactly-once is a combination of at-least-once and at-most-once delivery guarantee[1].

“Fig 1” shows increasing level of QoS, below.

****

*Figure 1 QoS levels*

In this paper, QoS level 2 will be examined and evaluated according to some of security properties on Scyther tool.

## 

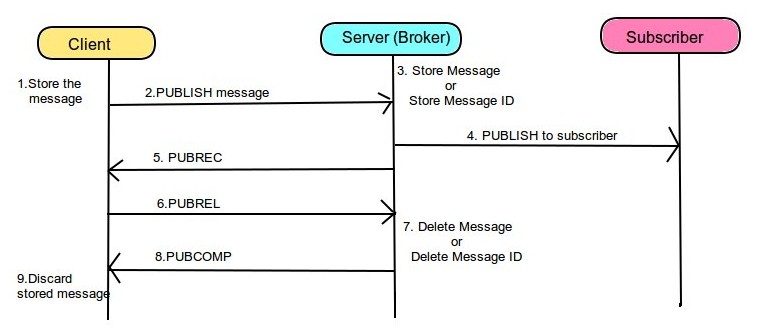
## A.1 QoS 2 level (Exactly once)

The third level of QoS can also be called “Assured Service”. This level of QoS transmits the message as two pairs of packets. The first pair is called PUBLİSH/PUBREC, while the second pair of messages is called PUBREC/PUBCOMP. There is absolute certainty that the message is transmitted at this QoS level. There is a maximum traffic at this QoS level transmitted in one go.

The Receiver(Server) gets a PUBLISH message from the client which is stored there. If the message has arrived at the receiver it will process the publish message accordingly and store the message or message ID on the server. The stored message is published to the subscriber. Server then gets an acknowledgment from Subscriber which will then be transmitted to the client as PUBREC.

The receiver will store a reference to the packet identifier until it has sent the PUBCOMP. This is important to avoid processing the message a second time. When the sender receives the PUBREC it can safely discard the initial publish, because it knows that the counter part has successfully received the message. It will store the PUBREC and respond with a PUBREL[2].

If the server gets the PUBREL, it can then delete the message or message ID information and answer with a PUBCOMP. After all, the originally stored message is no longer needed and can be discarded. A schematic that shows the operation of QoS level 2 is shown in figure 2.

****

*Figure 2 Operation of QoS 2*

When the flow is completed both parties can be sure that the message has been delivered and the sender is also informed[2].

Whenever a packet gets lost on the way, the sender is responsible for resending the last message after a reasonable amount of time. This is true when the sender is a MQTT client and also when a MQTT broker sends a message. The receiver has the responsibility to respond to each command message accordingly [2].

## Verification of QoS 2 level protocol on Scyther tool

Scyther[3] is a tool for the automatic verification of security protocols which was created by Cas Cremers. Scyther’s input language is loosely based on a C/Java-like syntax. The main purpose of the language is to describe protocols, which are defined by a set of roles.

In this paper, roles are generated as publisher broker and subscriber. If we need a give an example on Multi-Publisher and Multi-Subscriber, we can obtain the following information;

1. X and Y roles are publishers.
   * Mx and my are messages which must be delivered to publishers
2. PUBRACK, PUBREL and PUBCOMP are control messages. Showed in table 1

*Table 1 Control Packets*

|  |  |  |
| --- | --- | --- |
| **Control Packet** | **Direction of Flow** | **Description** |
| Publish | client→ server  client ←server | Publish message |
| Pubrec | client→ server  client ←server | Publish received |
| Pubrel | client→ server  client ←server | Publish release |
| Pubcomp | client→ server  client ←server | Publish complete |

1. Weakagree, Niagree, Nisynch claim events are representive of security properties. Descriptions of claim events in table 2 .

*Table 2 Security Properties*

|  |  |
| --- | --- |
| **Claim Events** | **Definitions** |
| Weekagree | A protocol guarantees to an initiator X weak agreement with another agent Y if, whenever X (acting as initiator) completes a run of the protocol, apparently with responder Y, then Y has previously been running the protocol, apparently with X.[5] |
| Niagree | A protocol guarantees to an initiator X non-injective agreement with a responder Y on a set of data items ds (where ds is a set of variables appearing in the protocol description) if, whenever X (acting as initiator) completes a run of the protocol, apparently with responder Y, then Y has previously been running the protocol apparently with X, Y was acting as responder in this run, and the two agents agreed on the data values corresponding to all the variables in ds.[5] |
| Nisynch | A protocol guarantees to an initiator X non-injective agreement with a responder Y on a set of data items ds if whenever X (acting as initiator) completes a run of the protocol apparently with responder Y, then Y has previously been running the protocol apparently with X, Y was acting as responder in this run, the two agents agreed on the data values corresponding to all the variables in ds, and each such run of X corresponds to a unique run of Y.[5] |

1. ack1X and ack1Y are acknowledgment informations.

## C. QoS 2 level (Exactly once)

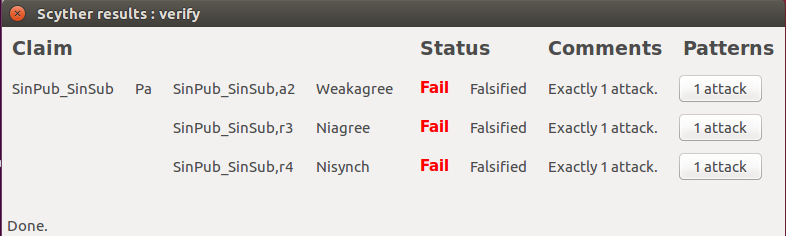
In this paper, 3 types of scenarios will be discussed. In the first scenario, only the relationship between the publisher and the subscriber will be examined. The second scenario examines the relationship between multi publishers and a single subscriber. The last scenario examines the relationship between multi publishers and multi subscribers. “Weakagree”, “Niagree” and “Nisynch” claim events are provided for each scenario.

## C.1. Single Subscriber and Single Publisher

(See spdl documentation in first part of Appendix)

As described above, firstly a message is published to the broker(receiver). The server sends the message which is received by the subscriber. The subscriber sends the information acknowledgment to the server. The server that received the acknowledgment can now deliver PUBREC to the client. Behind, Client answer to Server as PUBREL. Now that the server is sure the message has been is delivered,. it sends PUMCOMP in order to discard the message.

As shown below, verifying the Single Publisher-Single Subscriber QoS2 level protocol yields the following results as in Figure 3. Look at part 3 for the detailed verification result.



*Figure 3 Single Publisher-Single Subscriber Scyther result*

## C.2. Multi Publisher and Single Subscriber

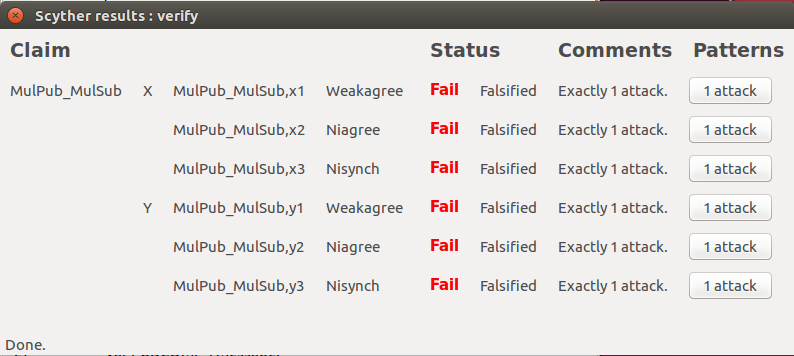
(See spdl documentation on second part of Appendix)

Considering multi publisher has the same scenario as the single subscriber-single publisher, the only difference is the message forwarded to Broker from different two roles. And then Broker fulfills the task and sends the messages to Subscriber.

Another thing that might be different is the Nonce type. It is used for defining of the message type in spdl code.

Nonce: A standard type that is often used and therefore defined inside the tool. [3]

Scyther result of Multi Publisher-Multi Subscriber is shown as in Fig 4. In this time, each role (each publisher) have their own Falsified result for each claim events. Look at part 3 for detailed verification result.



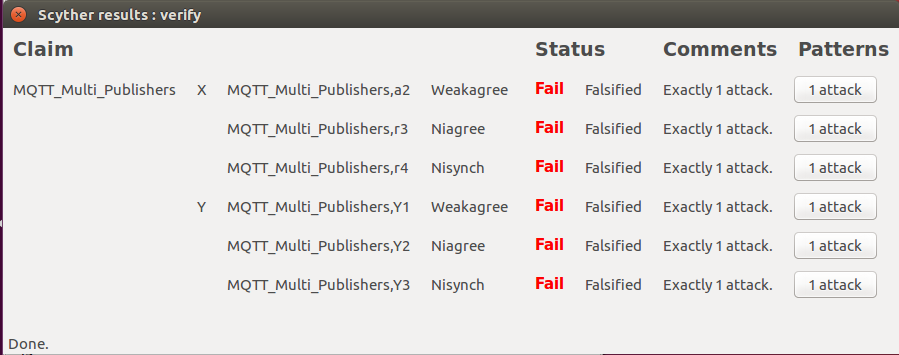
*Figure 4 Multi Publisher-Single Subscriber Scyther result (6)*

## C.3. Multi Publisher and Multi Subscriber

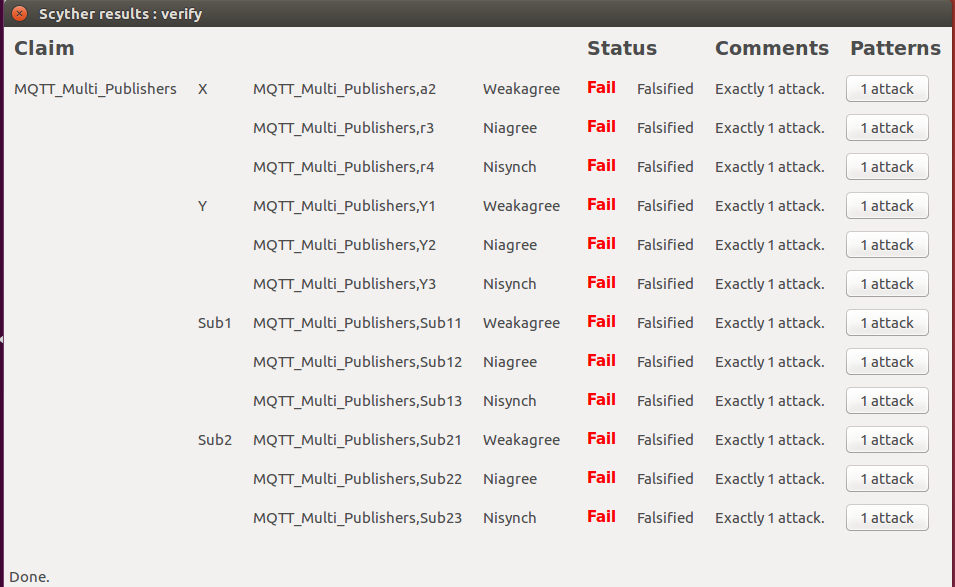
(See spdl documentation on third part of Appendix)

Multi Publishers and Multi Subscribers may seem like more complicated in the first view. However it’s still, the same scenario that repeats itself. This time, each the message of publishers must be gone each subscriber. According to arrived each message returns to as an acknowledgment information to Server. And, Server and publishers communicate amongst themselves with acknowledgment like PUBREL, PUBCOMP etc. for making sure arrival of messages.

Scyther result of Multi Subscriber - Multi Publisher is same with Multi Subscriber - Single Publisher as you can see below in Fig 5. The main point of similarity is about the usage of 2 roles of claim events control which is if there will be more than subscriber or publisher of claim event control, the result of verification can be changed according to the changes. For instance; If there would be claim events for each subscribers, result page, verification result might be as in Fig 6.



*Figure 5 Multi Publisher-Multi Subscriber Scyther result*



*Figure 6 Multi Publisher-Multi Subscriber Scyther*

Look at part 3 for detailed verification result.

# Attack Analysıs

Scyther Tool has its own verification system. After running verify of spdl code page, Scyther result page appears on the screen which is shown as possible attacks.

In Scyther results window, each claim event has their own single line. And their status is shown in the last two column called Status and Comments. Status column gives the actual result of the verification process: it will yield “Fail” when the claim is false, and “Ok” when the claim is correct. The “Comments” column explains results in the “Status” column. If it is written “OK, Verified” in Status Column, “No attacks” can be said in Comments for spdl code. There is a singular Comment for the correct result but there are multiple Comments for a status that yields “Fail”.

*Falsified Status Possibilities*

* At least X attack(s): There are certain attacks but might be more attacks.
* Exactly X attack(s): Number of attacks are certain known.
* No attacks within bounds: There is no attack within the bounded state-space, but it might be outside the bounded state-space.

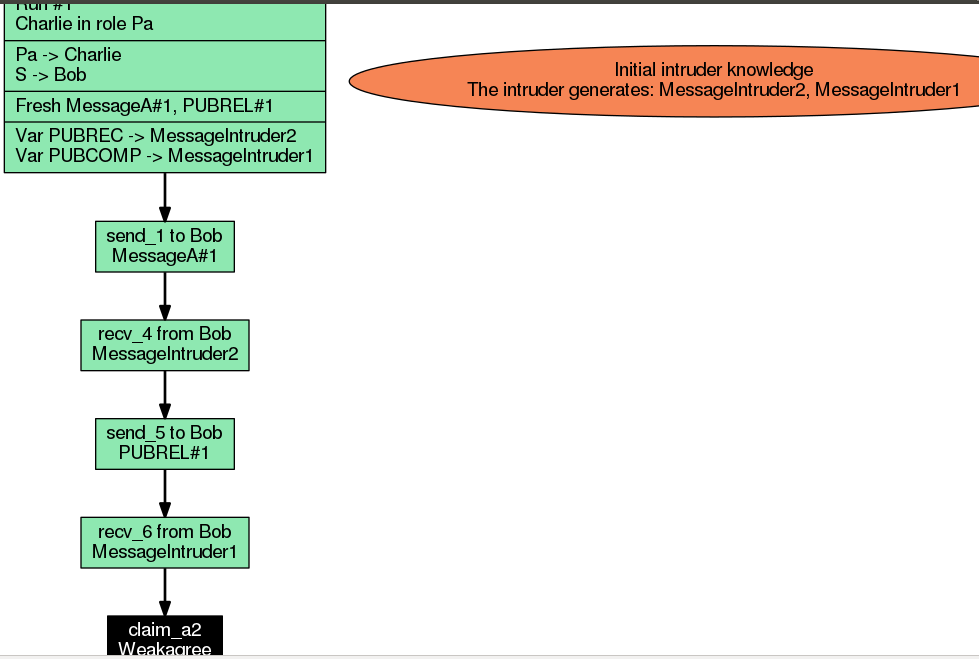
In this document, we have just “Exactly 1 Attack” comment for each QoS2 level code. Although all of the claims have an “Exactly 1 attack” result which occurs from different kinds of attacks, they actually originate from the same cause; value types. Generally, 2 value types are used for each QoS 2 level protocol scenario. These are “var” which means variable and “fresh” which means Freshly generated values.

* Variables[3] are rigid: after the first receive event in which they occur has been executed, they are assigned a value. This value cannot be changed afterwards.Variables must occur first in receive events: it is not allowed to use uninitialized variables in send events.
* Freshly generated values[3] are used in many security protocols for generating random values. They can be specified by declaring them inside a role definition using the “fresh” declaration.

As with the above explanation, “var” constant type is used for each “messages” usertype in first receive events(see codes in appendix). However, even though this usage is true as a definition , it indicates attacks in practice.

If we examine weakagree attack of role Pa in Single Publisher- Single Subscriber(Fig 7), we see that acknowledgment information which has a “var” type are detected as an intruder and as a result of weakagree claim event are not provided in QoS 2 level protocol.

Whereas, If first receiver is changed from type “var” to “fresh” in all QoS 2 level scenario, no attack is encountered.



*Figure 7 Weakagree attack*

# Conclusıon

In this paper, three different scenarios were created which are as follows Single Publisher-single Subscriber, Multi Publisher-Single Subscriber and Multi Publisher-Multi Subscriber. Then, the Quality of Services 2 level protocol was ensured in each scenario on Scyther Tool. QoS2 level is highest level protocol that also is combination of at-least-once and at-most-once delivery guarantee. The final step was seeing which of “Weakgree”,”Niagree”, “Nisynch” claim events were provided. In most cases, QoS2 level executives to the communication between Publisher to Subscribers without any intruders or malicious attacks, but in this paper we encountered with “exactly one attack” falsified which is same for every scenario. However, we can say that it is correct and dependable protocol practice in theory.

##### References

1. Peter R. Egli, “AN INTRODUCTION TO MQTT, A PROTOCOL FOR M2M AND IoT APPLICATIONS”, 2016, indigoo.com .
2. [www.hivemq.com](http://www.hivemq.com), “QTT Essentials Part 6: Quality of Service 0, 1 & 2M”.
3. C. Cremers, Scyther user manual, 2014.
4. B. Aziz, A. Arenas, B. Crispo, “Engineering secure IoT of things Systems”, 2016, p.160
5. Gavin Lowe. A hierarchy of authentication specifications. pages 31–43. IEEE Computer Society Press, 1997.

Appendix

1. Single Publisher - Single Subscriber

|  |
| --- |
| usertype Message;  usertype Acknowledgment;  protocol SinPub-SinSub(Pa, S, Sub) {  role Pa {  fresh MessageA : Message;  fresh PUBREL : Message;  var PUBCOMP : Message;  var PUBREC : Message;  send\_1(Pa, S, MessageA );  recv\_4 (S, Pa, PUBREC);  send\_5(Pa, S, PUBREL);  recv\_6(S, Pa ,PUBCOMP);  claim\_a1(Pa,Weakagree);  claim\_a2(Pa,Niagree);  claim\_a3(Pa,Nisynch);  }  role S {  var MessageA: Message;  var ack : Acknowledgment;  fresh PUBREC : Message;  var PUBREL : Message;  fresh PUBCOMP : Message;  recv\_1(Pa, S, MessageA);  send\_2(S, Sub, Pa, MessageA);  recv\_3(Sub,S, ack);  send\_4(S,Pa,PUBREC);  recv\_5(Pa, S, PUBREL);  send\_6(S,Pa, PUBCOMP);  }  role Sub {  var MessageA: Message;  fresh ack : Acknowledgment;  recv\_2(S, Sub, Pa, MessageA);  send\_3(Sub,S, ack);  }  } |

2. Multi Publishers - Single Subscriber

|  |
| --- |
| usertype Message;  usertype Acknowledgment;  protocol MulPub-MulSub(X, Y, S, Sub) {  role X {  const mx: Nonce;  var PUBREC : Message;  fresh PUBREL : Message;  var PUBCOMP : Message;  send\_1(X, S, mx );  recv\_5(S, X, PUBREC);  send\_7(X, S, PUBREL);  recv\_9(S,X, PUBCOMP);  claim\_x1(X,Weakagree);  claim\_x2(X,Niagree);  claim\_x3(X,Nisynch);  }    role Y {  const my: Nonce;  var PUBREC : Message;  fresh PUBREL : Message;  var PUBCOMP : Message;  send\_2(Y, S, my );  recv\_6(S, Y, PUBREC);  send\_8(Y, S, PUBREL);  recv\_10(S,Y, PUBCOMP);  claim\_y1(Y,Weakagree);  claim\_y2(Y,Niagree);  claim\_y3(Y,Nisynch);  }  role S {  var mx , my : Nonce;  var ackX , ackY : Acknowledgment;  fresh PUBREC : Message;  var PUBREL : Message;  fresh PUBCOMP : Message;  read\_1(X, S, mx );  read\_2(Y, S, my );  send\_3(S, Sub, X, Y, my, mx);  recv\_4(Sub,S, ackX, ackY);  send\_5(S, X, PUBREC);  send\_6(S, Y, PUBREC);  recv\_7(X, S, PUBREL);  recv\_8(Y, S, PUBREL);  send\_9(S,X, PUBCOMP);  send\_10(S,Y, PUBCOMP);  }  role Sub {  var mx , my : Nonce;  fresh ackX , ackY : Acknowledgment;  recv\_3(S, Sub, X, Y, my, mx);  send\_4(Sub, S, ackX, ackY);  }  } |

3. Multi Publishers - Multi Subscriber

|  |
| --- |
| usertype Message;  usertype Acknowledgment;  protocol MulPub-MulSub(X, Y, S, Sub1 , Sub2) {  role X {  const mx: Nonce;  var PUBREC : Message;  fresh PUBREL : Message;  var PUBCOMP : Message;  send\_1(X, S, mx, X);  recv\_5(S, X, PUBREC);  send\_6(X, S, PUBREL);  recv\_7(S,X, PUBCOMP);  claim\_a2(X,Weakagree);  claim\_r3(X,Niagree);  claim\_r4(X,Nisynch);  }    role Y {  const my: Nonce;  var PUBREC : Message;  fresh PUBREL : Message;  var PUBCOMP : Message;  send\_2(Y, S, my, Y);  recv\_10 (S, Y, PUBREC);  send\_11(Y, S, PUBREL);  recv\_12(S,Y, PUBCOMP);  claim\_a2(Y,Weakagree);  claim\_r3(Y,Niagree);  claim\_r4(Y,Nisynch);  }  role S {  var mx , my : Nonce;  var ack1X, ack1Y,ack2X, ack2Y : Acknowledgment;  fresh PUBREC : Message;  var PUBREL : Message;  fresh PUBCOMP : Message;  read\_1(X, S, mx, X );  read\_2(Y, S, my , Y);  send\_3(S, Sub1, X, Y, my, mx);  recv\_ 4(Sub1, S , ack1X,ack1Y);  send\_5 (S, X, PUBREC);  recv\_6(X, S, PUBREL);  send\_7(S,X, PUBCOMP);  send\_8(S, Sub2, X, Y, my, mx);  recv\_ 9(Sub2, S , ack2X,ack2Y);  send\_10(S, Y,PUBREC);  recv\_11(Y, S, PUBREL);  send\_12(S,Y, PUBCOMP);  }  role Sub1 {  var mx , my : Nonce;  fresh ack1X, ack1Y : Acknowledgment;  recv\_3(S, Sub1, X, Y, my, mx);  send\_ 4(Sub1, S , ack1X,ack1Y);  }  role Sub2 {  var mx , my : Nonce;  fresh ack2X, ack2Y : Acknowledgment;  recv\_8(S, Sub2, X, Y, my, mx);  send\_ 9(Sub2, S , ack2X,ack2Y);  }  } |