Analysing the PGM Protocol with Uppaal *

Béatrice Bérard Patricia Bouyer† Antoine Petit

LSV – UMR 8643 CNRS & ENS de Cachan,
61 av. du Prés. Wilson, F-94235 Cachan Cedex, France
{berard,bouyer,petit}@lsv.ens-cachan.fr

Abstract

Pragmatic General Multicast (PGM) is a reliable multicast protocol, designed to minimize both the probability of negative acknowledgements (NAK) implosion and the load of the network due to retransmissions of lost packets. This protocol was presented to the Internet Engineering Task Force as an open reference specification.

In this paper, we focus on the main reliability property which PGM intends to guarantee: *a receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data packet loss.*

We first propose a modelization of (a simplified version of) PGM *via* a network of timed automata. Using Uppaal model-checker, we then study the validity of the reliability property above, which turns out not to be always verified but to depend on the values of several parameters that we underscore.

1 Introduction

Since the introduction of timed automata (Alur and Dill (1990)), a lot of work has been devoted to both theoretical studies of timed models and practical issues for their analysis. Verification algorithms have been designed and implemented in so-called real-time model-checkers like HyTech (Henzinger et al. (1997)), Kronos (Bozga et al. (1998)) or Uppaal (Larsen et al. (1997)), with successful results for numerous case studies. In this paper, we propose the verification of two reliability properties for the multicast protocol PGM.

Reliable multicast protocols. Reliable multicast protocols are designed to enable distribution of information from multiple sources to multiple receivers, with reliability requirements. Examples of applications which may benefit from this technology include video broadcasts, data base replication or software downloads. Reliability in unicast protocols (like TCP) is usually achieved by positive acknowledgments (ACK) sent by the receiver to the source. Extending this principle to multicast protocols with a growing number of receivers may result in so-called ACK implosion. For this reason, the development of multicast protocols initially focused on eliminating ACKs, while keeping negative acknowledgements (NAK), invoked by receivers only when some packets are not received. However, multiple redundant NAKs can also be issued if packets are lost during periods of congestion. Besides, NAKs can be followed by redundant retransmissions.

PGM protocol. Pragmatic General Multicast (PGM) belongs to a second generation of reliable multicast protocols, designed to address the problems mentioned above: it is said to minimize both the probability of NAK implosion and the load of the network due to retransmissions of lost packets. The approach taken for buffer management resorts to timeouts at the source, with a new packet type called Source Path Message (SPM). This protocol was developed jointly by Cisco systems and TIBCO, and presented to the Internet Engineering Task Force as an open reference specification (Speakman et al. (2001)). It is currently supported as a technology preview, usually over IP, with which users may experiment, and it enters the longer-term standardization process.

*This work has been supported by the french project RNRT Calife.

†This work has been partly carried out while this author had a post-doctoral fellowship at BRICS, Aalborg University, Denmark

Contribution of the paper. In this paper, we focus on the main reliability property which PGM intends to guarantee (Speakman et al. (2001)): *a receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data packet loss.* Section 2 presents the main features of PGM. In Sections 3 and 4, we describe how a simplified model of the protocol is built, *via* a network of timed automata. Section 5 is devoted to a detailed presentation of the verification process and Section 6 concludes the paper.

2 Description of PGM

We first give a brief description of the protocol, with a single source, as proposed in (Speakman et al. (2001)). Since we are only interested in reliability, we omit all mechanisms related to minimization of the load.

The source multicasts sequenced data packets called ODATA (for Original Data), within a transmit window, at a given rate. Those packets are transmitted through network elements, along some path of a distribution tree. If a receiver detects a missing packet from the expected sequence, it repeatedly unicasts a negative acknowledgement (NAK) to the last element on the path. This network element forwards the NAK to the source using the reverse path, and multicasts a NAK confirmation (NCF). The receiver stops sending the NAK upon reception of the NCF. The same operation is repeated in turn at each level of the path toward the source, including the source itself. Repairs (RDATA) must then be provided by the source or by a Designated Local Repairer (DLR). Since the reliability of the protocol mainly depends on the NAK transmissions, PGM defines a network-layer hop-by-hop procedure for reliable forwarding. A similar method is used for NCF multicasting.
In addition to this basic data transfer operation, SPMs (Source Path Messages) are sent by the source at a given rate, thus periodically interleaved with ODATA. Their purpose is twofold:

1. maintaining up-to-date neighbour information, to ensure that the return path for the NAKs is exactly the reverse of the forward path followed by ODATA and

2. delivering information on transmit resources to the receivers.

The architecture of a network on which PGM can be used is depicted on figure 1.

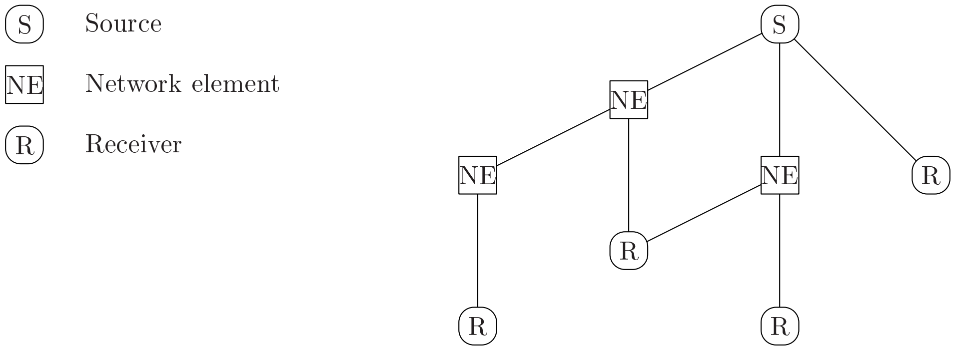


Figure 1: An overview of PGM

Transmit window. More precisely, assuming a constant data packet size, ODATA packets are ordered by the source in unit increments from a circular sequence number space from 0 to $2^{32}-1$, and buffer management relies on the transmit window maintained by the source:

- the left edge of this window, `txw_trail`, is the number of the oldest data packet available for repair,

- its right edge, `txw_lead`, is the number of the most recent data packet transmitted.

The window is considered empty if $\texttt{txw\_trail} = \texttt{txw\_lead} + 1$ and its size is bounded by $2^{16} - 1$. There is no fixed strategy for the source to advance the trailing edge of the window. The edges of the transmit window are contained in SPMs. A receiver also maintains a receive window, with edges `rxw_trail` and `rxw_lead`, which evolves according to the informations received from the source, either by data packets or SPMs.

Rates and priorities. The source must strictly prioritize sending of pending NCFs first, pending SPMs second, and only send ODATA or RDATA when no NCFs or SPMs are pending. The priority of RDATA versus ODATA is application dependent, with a possible sharing strategy. Before the source multicasts some RDATA upon reception of a NAK, there may be some bounded delay to wait for other NAKs.
There are two types of SPMs: in the presence of O/RDATA, ambient SPMs are transmitted by the source at a regular rate, while heartbeat SPMs are transmitted in the absence of data at a decaying rate, in order to maintain receive windows and assist early detection of lost data.

3 Modelling PGM with timed automata

In this section, we explain how our model of the protocol is built. Since the protocol involves timing constraints, the choice of a timed model-checker is mandatory. Besides, many discrete variables appear, with an array-like organization. This leads to the tool Uppaal (Bengtsson et al. (1998)), which offers a compact description language for this type of variables, thus making modelling easier.

3.1 Modelling with Uppaal

A complete presentation of the model handled by Uppaal can be found in many papers or surveys (see for example (Larsen et al. (1997); Bengtsson et al. (1998); Amnell et al. (2001); Bérard et al. (2001))). Timed automata in Uppaal are a modified version of the original ones from (Alur and Dill (1994)). We briefly recall here in an informal way the main characteristics of the model, illustrated in Section 4 below (see Remark 4.1).

In Uppaal, a system consists of a collection of timed automata, with binary synchronization: two components synchronize through channels with a sender/receiver syntax. For instance, on a given channel `c`, a *sender* emits a signal denoted by `c!` and a *receiver* synchronizes with the sender by the corresponding reception denoted by `c?`. These two actions are called *complementary actions*. A timed automaton is a finite structure handling a finite set of variables. These variables are either *clocks*, which evolve synchronously with time, or bounded discrete integer variables. A location in an automaton can be labelled by clock conditions, called *invariants*, which must be satisfied as long as time elapses in this location. A *transition* of the automaton is decorated by three types of labels:

- a *guard* expressing a condition on the values of the variables, which must be satisfied for the transition to be fired,
- a synchronization *label* of the form `c!` or `c?`,
- a *clock reset* and an *update* of integer variables.

Note that each type of label is optional, an absence of synchronization meaning that the automaton performs an internal action.

A (global) configuration is of the form (ℓ, v) where ℓ is a location vector (indicating the current state in each component of the timed automata network) and v is a valuation of the variables, that is a function which assigns to each clock a real value and to each discrete variable an integer value. An execution in the network starts in initial locations of the different components with all the clocks and variables set to zero. The semantics of this model is expressed by moves between the configurations. Three types of moves can occur in the system: delay moves, internal moves and synchronized moves.

Delaying. As long as no invariant is violated in the current locations, time may progress, without changing the location vector and the values of the discrete variables. Clock values increase by the time elapsed.

Performing an internal action. If an internal action is enabled in a component (*i.e.* a transition decorated by no synchronization label), then the component can perform this internal action, without synchronizing with another component. The configuration is changed following the same rules as for the synchronization described below.

Synchronizing. If, in the network, two complementary actions are enabled in two different components (in particular, both guards must be verified by the current valuation), then these two components can synchronize. The location vector is changed in a natural way and the clock and variable values are changed following the clock resets and the updates of variables.
Note that the channels in UPPAAL are synchronizing channels but not communicating channels, in the sense that no information is transmitted. To model data transfer through a channel, we need global variables. For example, sending some information on a channel `c` is simulated as follows: we define two global variables `info_to_be_sent` and `info_received`. The sender has a transition labelled by `c!` whereas the receiver has a transition labelled by `c?` with update `info_received := info_to_be_sent`. The value of the variable `info_to_be_sent` is then 'transmitted' on channel `c`.

Another feature of UPPAAL will be very useful in our modelling: locations can be decorated by some special label called *committed* (in the next figures, committed locations are labelled by a `c`). This notion is useful to model atomicity of transition sequences in some given component. If a current location is labelled as committed, then no delay is enabled before the next non-delaying move is performed and this next non-delaying evolution must involve the component to which belongs the committed location. In that way, two transitions linked through a committed location will be taken sequentially without delay. Note that this mechanism reduces the non-determinism in the parallel composition of the different components. Therefore, it will also lead to a reduction of the set of reachable states of the system.

3.2 A simplified model

As is always the case, our model is an abstraction of the protocol, leaving out details which do not influence the properties of the model we want to check. Its size must also be reduced in order to obtain reasonable performances in terms of memory consumption and automatic verification time. However, for an evident correctness criterium, the model has to be sufficiently detailed for catching potential errors in the protocol. We will see that even with strong reductions, we exhibit some weaknesses, inherent in the protocol.

We consider a fixed topology for the network, which corresponds exactly to the distribution tree, so that routing information is not necessary. While transmission errors due to routing deficiencies can occur, they would not directly result from the protocol. Therefore, fixing the topology is a suitable abstraction with respect to the properties to check. As explained in Section 2, specific mechanisms are designed to ensure the reliability of NAK forwarding. Therefore, we may assume no loss of NAKs, so that a single NAK may be issued by a receiver and sent directly to the source, and NCFs are omitted. We also assume that no additional DLR are created, so that only the source can send repairs. Finally, we consider a constant rate for all SPMs, even in the absence of data packets (heartbeat mode) and constant data packet size. From this hypothesis and the fact that we are not interested in the real content of the data transmitted, we represent data by their index. Hence, it will not be possible to detect corruption of messages. However, we are mostly interested in questions related to message losses, so that the simplification is correct in this sense. In the same way, a NAK simply consists in the index of the data detected as lost. Figure 2 shows a minimal configuration and summarizes the simplifications proposed above.
These simplifications concern either the topology of the network or characteristics of the transmission, which strengthen the reliability of the protocol (e.g. no loss of NAK). Therefore, we

cannot hope to prove correctness properties. On the other hand, if we can find execution traces for specific misbehaviors of the model, then we can be sure that these problems appear in the real protocol.

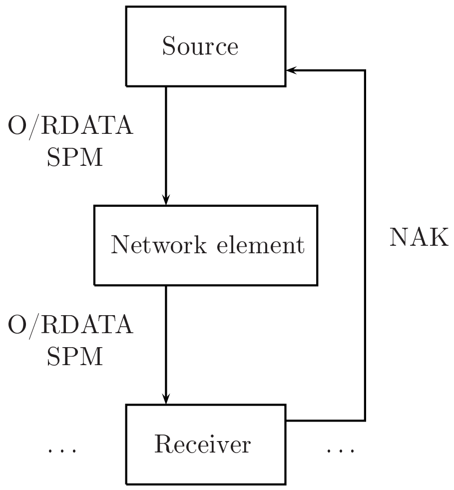


Figure 2: Scheme for the simplifications

The global model of the PGM protocol is built in a compositional way from models for the different components. Each source or network element (called 'transmission') is represented by a timed automaton, while each receiver is composed of two automata running in an asynchronous way. The first one is designed to receive data and SPMs whereas the other detects losses and sends NAKs. We obtain this way a set of $2 + 2n$ timed automata, for n receivers.

A general scheme of our model is depicted in figure 3, with the following conventions: communicating channels are drawn with edges between components, the starting point of the edge being the sender and the target of the edge being the receiver (with respect to the channel). Global variables used to transfer data on the channels are written in brackets. The main discrete variables are written in `typewriter` style in each component and square brackets like `[]` indicate arrays. Variables used as indexes or which are not really important are omitted. Clocks are written in *italic*.

Communications between the components are modelled as follows. The source communicates with a network element by transmitting O/RDATA and SPMs, *via* the two channels 'src_ne_data' and 'src_ne_spm' respectively. As explained before, global variables are used for data transfers: for the first channel, the variable 'src_sqn' represents the number of the current data transmitted. For the second channel, we use two global variables, 'src_trail' and 'src_lead' for the current state of the transmit window. In turn, the network element can send to each receiver the same information, thus similar channels and global variables are used. A receiver communicates with the source by sending NAKs, which is coded in the same way.

4 The components of the model

We now describe each component of the model as a timed automaton. To simplify the descriptions and the figures, we assume that there is only one network element (therefore the source sends messages to this element only) and two receivers (and thus the network element sends messages to both receivers and can get NAKs from any of them).

4.1 The source

The three tasks of the source are:

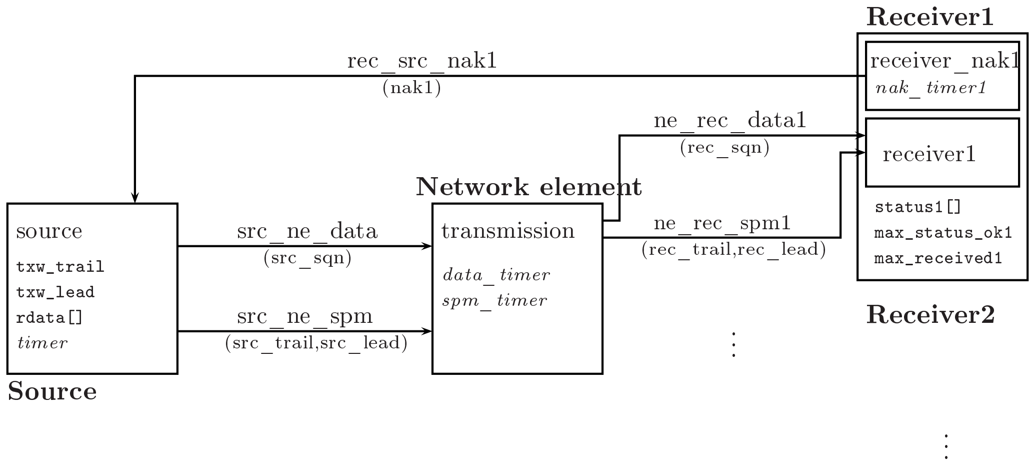
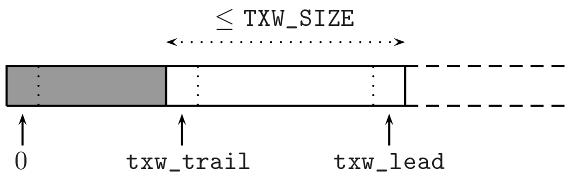


Figure 3: An overview of our PGM model

- sending SPMs, to give information on the transmit window,
- sending O/RDATA, which involves an update of the transmit window,
- receiving NAKs, which updates the set of pending RDATA.

The transmit window is modelled using two variables, `txw_trail` and `txw_lead`, which correspond respectively to the left edge of the window, that is the number of the oldest data packet available for repair and to the right edge of the window, that is the number of the most recent data packet transmitted. This window is handled in the following way: it has a maximal size, represented by the parameter `TXW_SIZE` and when the window is full, that is when `txw_lead - txw_trail` = `TXW_SIZE`, all the RDATA in stand-by are sent and the transmit window is advanced, which means that `txw_trail` is incremented of the parameter `TXW_ADVANCE`. A configuration of the transmit window can be depicted in the following way:



The set of pending RDATA is represented by the array `rdata`. The source is modelled by the automaton in figure 4, with its three tasks.

When the source receives a NAK, it has to keep its value in the table `rdata`. Sending SPMs must be done periodically but it does not change the data structures belonging to the source. Sending data is the most complicated part of the source: if the transmit window is not full (*i.e.* `txw_lead - txw_trail` < `TXW_SIZE`), then the source can send ODATA to the network (the periodicity of sending data is parameterized by `MIN_DATA_PERIOD` and is checked thanks to the clock `timer`) ; if the transmit window is full (*i.e.* if `txw_lead - txw_trail` = `TXW_SIZE`), then the source can not send ODATA, it must first send all pending RDATA. If there are no more pending RDATA, then the transmit window is advanced before sending ODATA. Sending RDATA involves updating the table `rdata`, which is done in the right part of the automaton.

Remark. The main features of timed automata informally introduced in Section 3.1 are illustrated in the figure above. The left-most transition at the bottom is labelled by a condition `rdata[0] == -1`, a synchronization label `rec_src_nak?` and an update of the discrete variables `rdata[0]` and `nb_rdata`. The condition `rdata[0] == -1` means that there is no pending RDATA. The transition can be taken as soon as the condition is met and as soon as some other component of the whole system can synchronize on the label `rec_src_nak?`, *i.e.* as soon as a transition of some other component is labelled by `rec_src_nak!`. In the 'Sending SPMs' part, a location is labelled as

'Committed': this is because the two updates from the ingoing transition and the synchronization `src_ne_spm!` must be atomic. Besides, note that in the part 'Receiving NAKs' of the automaton in figure 4, a UPPAAL syntactic short-cut is used several times: `value:= (cond?value1:value2)` means that if `cond` is true, then `value` is set to `value1`, otherwise, it is set to `value2`.

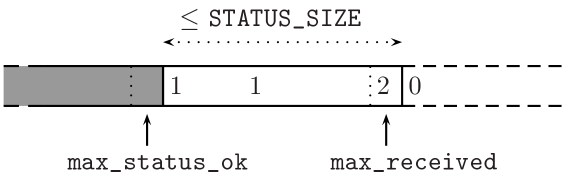
4.2 A receiver

The task of a receiver is twofold: it has to receive data (updating its receive window), and to send NAKs in case of loss detection. These two tasks are independent, thus the model for a receiver consists of two timed automata running in parallel without synchronization. Figures 5 and 6 represent the two parts of a receiver.

The main data structure associated with a receiver is the receive window. While the original protocol adds some structure to take into account losses of non contiguous data, we use a simplified structure replacing both mechanisms: we use a circular array called `status` whose size is set by the parameter `STATUS_SIZE`. Two numbers of data are kept in mind, the first one, `max_received`, represents the highest number of a received data whereas the second one, `max_status_ok` represents the highest number for which the status is known by the receiver: for all data indexed by some integer less than (or equal to) `max_status_ok`, the receiver knows if the data has been received or if it will never be received. Each entry of the table `status` is in one of the following state:

- 0 if the entry is empty, meaning that the receiver has no information on the corresponding data
- 1 if the receiver has to send a NAK for the corresponding data
- 2 if the corresponding data has been received

The receive window can be depicted in the following way:



Upon reception of a data or a SPM, the receiver has to update the `status` inside the receive window, which is done by the automaton on figure 5. Again transitions are linked through a committed location, since their actions are to be performed sequentially and immediately. Note that a variable `test` was added in this automaton, to help in the verification process (see Section 5.1).

The second part of the receiver is designed to send NAKs. The corresponding automaton is represented in figure 6. The array `status` is glanced through, and when a missing data is detected (the state of the corresponding entry in the table is 1) the receiver must send a NAK. Note that a loop is the only way to achieve searching through an array. Moreover, using another timed model-checker would have made this more difficult on a syntactic point of view. The NAKs must be sent each `NAK_PERIOD` units of time, which is done with the clock `nak_timer`. When no NAK needs to be sent, the upper loop on the initial state resets the clock `nak_timer`, waiting for the detection of new lost data.

4.3 A network element

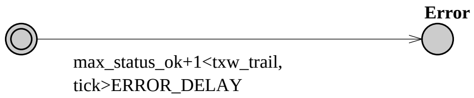
The task of a network element is to transmit data and SPMs to the receivers. Recall that the specification requires sending SPMs interleaved with data, and additional SPMs when no data is sent. We use the network element to simulate data losses, transmission delays and SPMs rate (if no data is sent). A network element for two receivers is represented on figure 7.

The right part of the automaton (above the diagonal transition) aims at transmitting data. The curved transition on the right represents the data loss (the part transmitting data to the receivers is skipped). As said in section 3.1, since binary synchronization is used in UPPAAL, the network element cannot broadcast data simultaneously to the two receivers. The solution we did choose is to send the data to the first receiver (the transmission delay is tested at this point) and then, using a committed location, the same data is sent right afterward to the second receiver. The left part of the automaton (below the diagonal transition) aims at sending SPMs, in the same way than sending data (apart from the fact that SPMs cannot be lost). This automaton enforces in addition that after a data is sent to the receivers, a SPM has also to be sent. The diagonal transition is used when the source only sends an SPM (this must be done at a given rate parameterized by `MIN_SPM_PERIOD`) without sending a data.

5 Verification of PGM properties

5.1 The two properties we will check

The first property we want to verify is the one specified in (Speakman et al. (2001)): *For each data, each receiver knows if it did receive the data or if it will never receive it.* We call it **Loss-info**. To check this property, we need to add an external observer, also called *test automaton*, to the model previously described. The general structure of an observer contains an error state, reached when the property is violated. In our case, the test automaton (drawn on the next figure) has to compare the receive window with the transmit window and the property **Loss-info** is not met only if the state `Error` is reachable.



Indeed, the data indexed by `max_status_ok + 1` is the first whose status is not known by the receiver. If this data is no more in the transmit then the receiver will no more receive this data, this last one will thus be lost. As the trailing edge of the transmit window is `txw_trail`, this condition expresses as `max_status_ok + 1 < txw_trail`, which is precisely the condition for reaching state `Error`. In addition, a clock `tick` and a new parameter `ERROR_DELAY` have been added to the receiver currently tested. The clock is reset each time the receiver updates its receive window. In that way we can compute the reaction time of the network, represented by the parameter `ERROR_DELAY`: it is the maximal time for a data to go from the source to a receiver. Finally, the observer can reach state `Error` only if the previous condition on the windows is met and a sufficient time has elapsed since the last update of the receive window.

The second property we are interested in is the loss of data. As explained in the PGM specification document (Speakman et al. (2001)), it is hopeless in general to have no such loss. Nevertheless, we will investigate some hypotheses, under which this very strong property is verified. We define the property **No-loss** as: *Each data which is detected as lost is eventually repaired.*
To express this property, we do not need any more an external observer. The negation of this property directly expresses the fact that, for a given receiver, no state where the variable `test` is equal to `1` is reachable.
Indeed, the value of the variable `test` in the automaton `receiver` becomes 1 if, when updating the array `status`, `max_status_ok` has to be increased whereas the missing data corresponding to the index `max_status_ok`+1 has never been received.

5.2 Experiments

We used the model-checking tool UPPAAL[1] to implement the model we just described. We ran our experiments on a Sun Ultra 220R with 2x450MHz CPU and 2048 MB RAM (with the OS Solaris 8).

Experiments will make the values of some parameters vary. Parameters are constants of the system which are not fixed *a priori*. The first two parameters are for the topology of the network (number of receivers and of network elements), they will not change much in our experiments. There are 6 parameters for the sizes of the data structures used in the model (bounds for integers, array sizes, etc...), with fixed values for all the experiments (see table 1). Note that we had to choose small values to take into account memory consumption problems. The 7 other parameters are time parameters, which aim at controlling the "traffic" on the network. A question mark in table 1 indicates that their value will change along the experiments.

Global parameters of the protocol		
Number of receivers		1 or 2
Number of network elements		1
Size parameters		
Number of data the source can send	`MAX_NB_DATA`	10
Max. number of RDATA in stand-by	`MAX_NB_RDATA`	2
Max. number of lost data at some point	`MAX_NB_ERROR`	2
Receive window size	`STATUS_SIZE`	3
Transmit window size	`TXW_SIZE`	5
Transmit window advance	`TXW_ADVANCE`	2
Time parameters		
Transmission delay for SPM	`SPM_DELAY`	?
Lowest bound for the SPM sending period	`MIN_SPM_PERIOD`	?
Highest bound for the SPM sending period	`MAX_SPM_PERIOD`	?
Lowest bound for the data transmission delay	`MIN_DATA_DELAY`	?
Highest bound for the data transission delay	`MAX_DATA_DELAY`	?
Data sending period	`MIN_DATA_PERIOD`	?
NAK sending period	`NAK_PERIOD`	?

Table 1: Parameters of the experiments

5.2.1 Verification of the property Loss-info.

We present the experimental results in the case of one receiver.[2] The model has 960 control states (in the synchronized product), 5 clocks and 25 bounded variables. Table 2 summarizes our results for the property **Loss-info** for one receiver, with time parameter values fixed as indicated below:

`SPM_DELAY`	1
`MIN_DATA_DELAY`	2
`MAX_DATA_DELAY`	3
`MIN_SPM_PERIOD`	3
`MAX_SPM_PERIOD`	5
`MIN_DATA_PERIOD`	1
`NAK_PERIOD`	2

The property checked is expressed by the formula `E<>(obs.Error)` in the UPPAAL syntax. The answer **True** says that this property holds, which means that the state `Error` of automaton `obs` is reachable. Thus, even after having waited `ERROR_DELAY` units of time, the receiver can make a

[1]More precisely, we used the version 3.2.4 of UPPAAL2k, see `http://www.uppaal.com`

[2]After 28 hours of computation, we did stop the verification process for two receivers on a Sun Ultra 420R with 4096 MB RAM.

ERROR_DELAY	**5**	**6**	**7**	**8**	**9**	**9**	**10**	**11**	**12**
Answer	**True**				**False**				
Options	Breadth-first[a] Active-clock reduction[b]					Breadth-first Active-clock reduction Convex-hull approximation[c]			
Time (in s) [3] Memory (in KB)	193 112464	898 445232	911 449776	1007 480728	1753 771280	550 222312	556 222312	552 222312	555 222312

[a]The graph searching order is breadth-first.
[b]Clocks which are not useful are detected and eliminated.
[c]At each step of the computation, if two reachable zones have been computed for the same location, the tool keeps in mind the smallest zone containing both zones. Using such an overapproximation, if the model-checker finds that a state is reachable, it is possible that it does a mistake. That is the reason why this approximation can only be used in case of negative answer.

Table 2: Results for the property **Loss-info**

mistake and can think that it will be able to receive a repair for some data, whereas it is impossible because the data is no more in the transmit window of the source. It thus means that the property **Loss-info** is *not* met.

Analyzing table 2 shows that 9 is the limit-value (answer **False** for 9 and after and **True** before). It appears thus as the transmit reaction of the network, *i.e.* the time for the receiver to get the information about the transmit window. As a conclusion, a receiver always knows in a bounded time that he will never receive a lost data. The 'reaction time' depends on all the time parameters of the network (transmission delays of data, SPMs, frequency of sending SPMs,...).

5.2.2 Verification of the property No-loss.

We present the experimental results in the case of two receivers. The model has 17280 control states, 5 clocks, 35 bounded discrete variables. The results we present assume that the following parameters are fixed.

SPM_DELAY	1
MIN_DATA_DELAY	2
MAX_DATA_DELAY	3
MIN_SPM_PERIOD	3
MAX_SPM_PERIOD	5
NAK_PERIOD	3

Our results are summarized in the table 3.

MIN_DATA_PERIOD	**1**	**2**	**3**	**4**
Answer of UPPAAL	**True**	**True**	**False**	**False**
Options	Breadth-first Active-clock reduction		Breadth-first Active-clock reduction Convex-hull approximation	
Time (in s) Memory (in KB)	4743 1239832	5775 1402200	918 630608	9 39 597440

Table 3: Results for the property **No-loss**

The value **True** in the table interprets as 'the model verifies the formula `E<>(receiver1.test==1 or receiver2.test==1)`'. Indeed, following UPPAAL syntax, this formula expresses that there exists a reachable state in which the variable `test` of `receiver1` or `receiver2` is equal to `1`. It thus corresponds to the negation of property **No-loss**.

Comments on the results. As we explained above, and from the comments of the specification document, it is far from being a surprise that for some values the property **No-loss** is not verified. The aim of PGM protocol is indeed not to ensure that data are never lost but rather than the status of all data is known for each receiver. Our experiment confirms that if the frequency of sending data (represented by the parameter `MIN_DATA_PERIOD`) is too high, the property **No-loss** is not verified.

But UPPAAL also gives execution traces for which the property is not verified. Such a trace corresponds to a scenario where the source sends an RDATA to the network. This RDATA is erased from the table `rdata`. If this same RDATA is lost once more by the network, it is possible that the source has erased the data from its knowledge by advancing the transmit window because it seems that, by now, all receivers have obtained the repaired data. However, this is not the case, and the next NAK from a receiver informing the source that this data has been lost will arrive too late to the source.

Note that, in a perhaps unexpected way, our experiments (see Table 3) also show that, if the sending of data is slow, then data are never lost. This very strong property is thus verified for some values of the parameters even if the PGM protocol does not intend to guarantee it.

We did also test a stronger property which is
`A[](receiver1.test==0 and (source.nb_rdata ==0 or source.live<LIVENESS))`
where:

- `nb_rdata` is the number of standing-by RDATA up to the source,
- `live` is a new clock belonging to the source, which is reset each time a data is sent and
- `LIVENESS` is a parameter which measures the liveness of the sending of data.

This property expresses that in each reachable configuration, there is no detection of lost data (variable `test` equal to 0) and either there is no waiting RDATA or the network is *sleeping* since less than `LIVENESS` units of time. We did test this property for several values of `LIVENESS` and for several values of `MIN_DATA_PERIOD` (like for the test of property **No-loss**) and the result is always that this property is never met. It means thus that even for values of `MIN_DATA_PERIOD` for which no data are lost, if the network is not alive (in the sense that no data is sent), then some RDATA can wait an infinite amount of time without being sent and, in that way, even if the source does not erase the corresponding data from the transmit window, the receiver does not receive a repaired data.

6 Conclusion

In this work, we proposed a model based on timed automata to represent (a simplified version of) the Pragmatic General Multicast (PGM) protocol. This model has been designed from the specification provided by (Speakman et al. (2001)), in a compositional way, with a timed automaton representing each of the actors (source, network elements or receivers) of the protocol. We deeply use the main features of timed automata, in particular to model the sending/receiving of messages and to take into account timing constraints, with clocks, bounded variables, tables, channels, global variables as well as local variables and committed locations.

All the implementation task used the tool UPPAAL whose GUI and simulation module turned out to make the modelling phase easier.
The verification tasks have been performed with the model-checker module of UPPAAL. An interesting and useful feature of this module is to provide, in case of failure of the tested property, a trace of an execution for which the property is not verified. We used this feature to give counterexamples for the values of the parameters making the studied properties false.

Note that even if our model represents only a (very) simplified version of PGM, it allows us to prove that the main reliability property which PGM intends to guarantee *'a receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data packet loss'* is verified only if there is a sufficient reaction delay before the receiver gets a good knowledge of the transmit window.

Finally, remark that, even with this simple model, we have been confronted with big memory consumption problems. Therefore it is probably hopeless to verify with a model-checker a more elaborate version of the protocol. But our model can be used for several other interesting testings, concerning in particular the parameters that have been fixed within our experiments. Since the results appear to depend on the relative frequencies of sending and transmitting data and NAKs, we should look for a dependency relation between the corresponding values. It is thus worthwhile to multiply the experiments in order to have a better understanding of PGM.

References

Alur, R. and Dill, D. *Automata for modeling real-time systems.* In Proceedings of the 17th International Colloquium on Automata, Languages, and Programming (ICALP'90), vol. 443 of Lecture Notes in Computer Science, pp. 322–335. Springer-Verlag, 1990.

Alur, R. and Dill, D.. A theory of timed automata. *Theoretical Computer Science*, 126(2): 183–235, 1994.

Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, O., Pettersson, P., Weise, C., and Yi, W. Uppaal– *now, next, and future.* In Proceedings of the Modelling and Verification of Parallel Processes (Movep2k), vol. 2067 of Lecture Notes in Computer Science, pp. 99–124. Springer-Verlag, 2001.

Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W., and Weise, C. *New generation of* Uppaal. In Proceedings of the International Workshop on Software Tools for Technology Transfer, 1998.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., and Schnoebelen, Ph. *Systems and Software Verification - Model-Checking Techniques and Tools.* Springer-Verlag, 2001.

Bozga M., Daws, C., Maler, O., Olivero, A., Tripakis, S., and Yovine, S. Kronos*: a model-checking tool for real-time systems.* In Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), vol. 1427 of Lecture Notes in Computer Science, pp. 546–550. Springer-Verlag, 1998.

Henzinger, T.A., Ho, P.-H., and Wong-Toi, H. HyTech: A model-checker for hybrid systems. *Journal of Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

Larsen, K.G., Pettersson, P., and Yi, W. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

Speakman, T. et al. *PGM Specification*, 2001. Internet draft of the IETF (Internet Engineering Task Force), 115 pages.

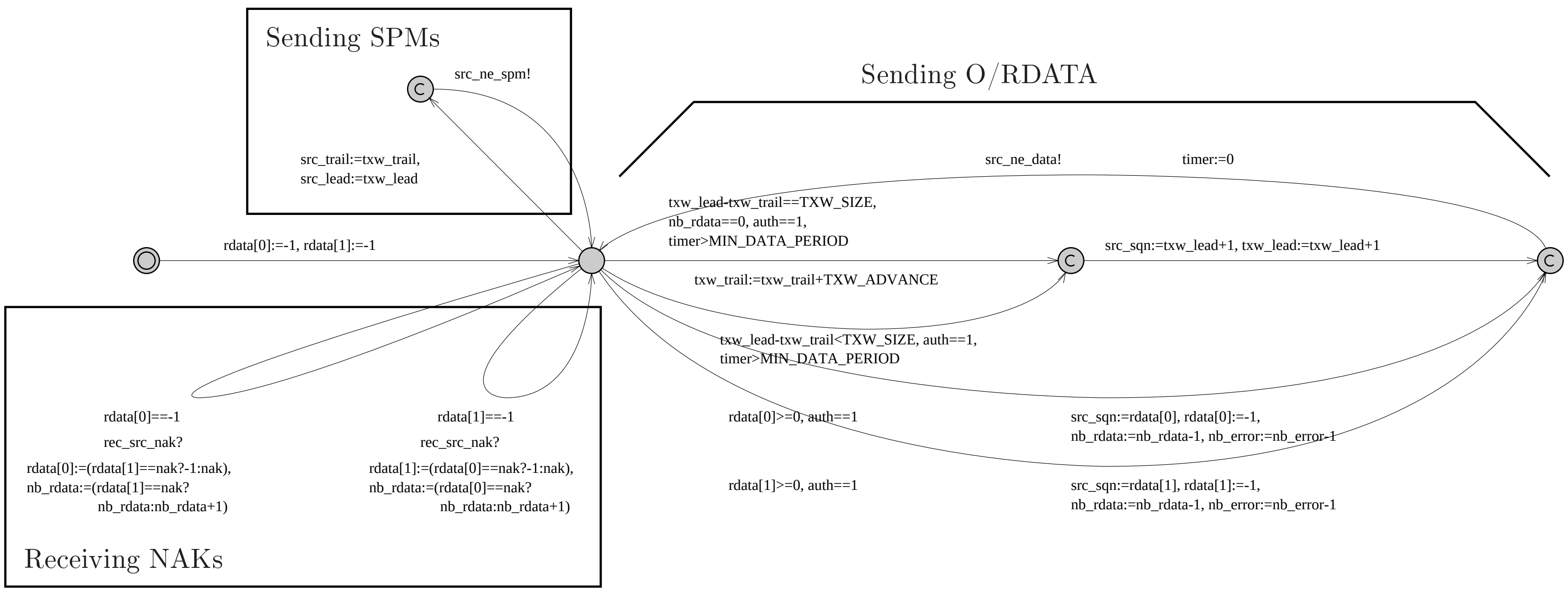


Figure 4: Model for the source

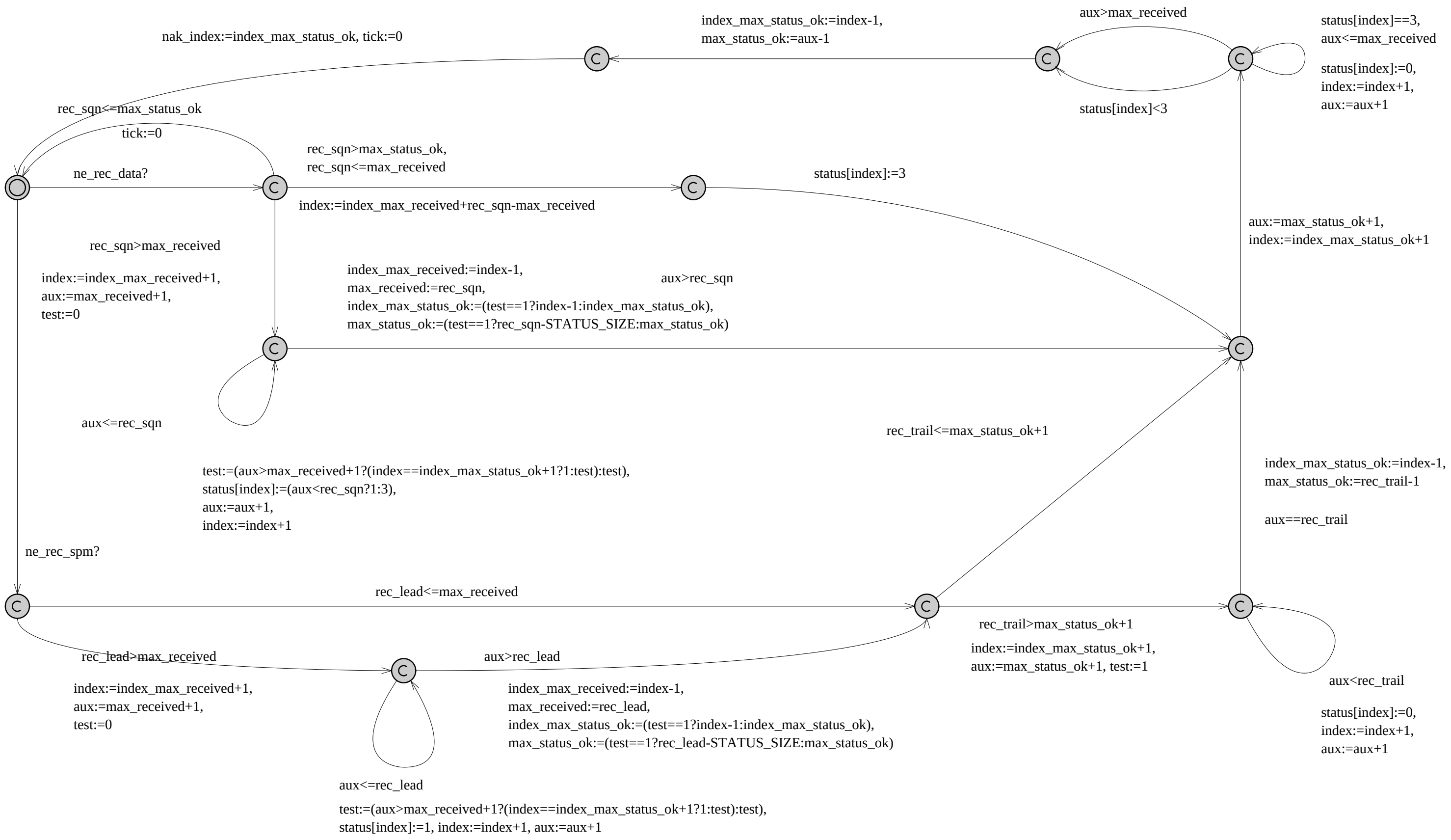


Figure 5: Part of a receiver receiving data and SPMs

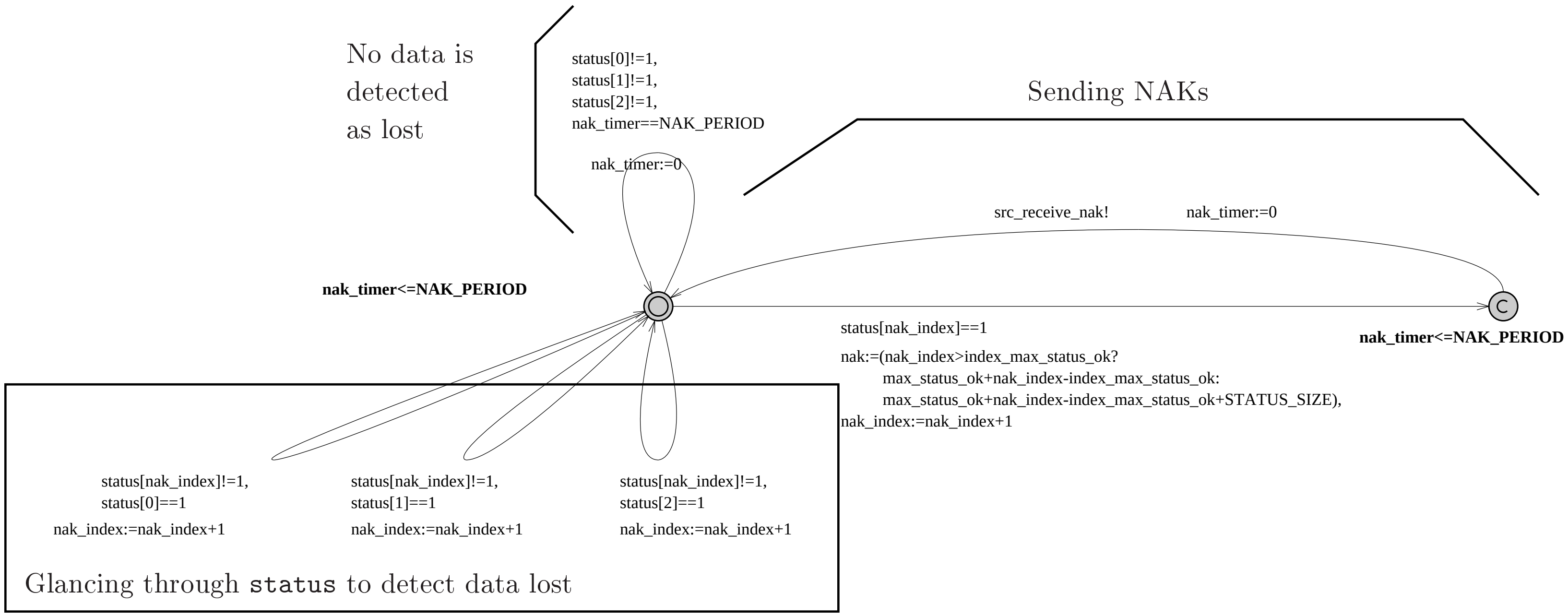


Figure 6: Part of the receiver sending NAKs

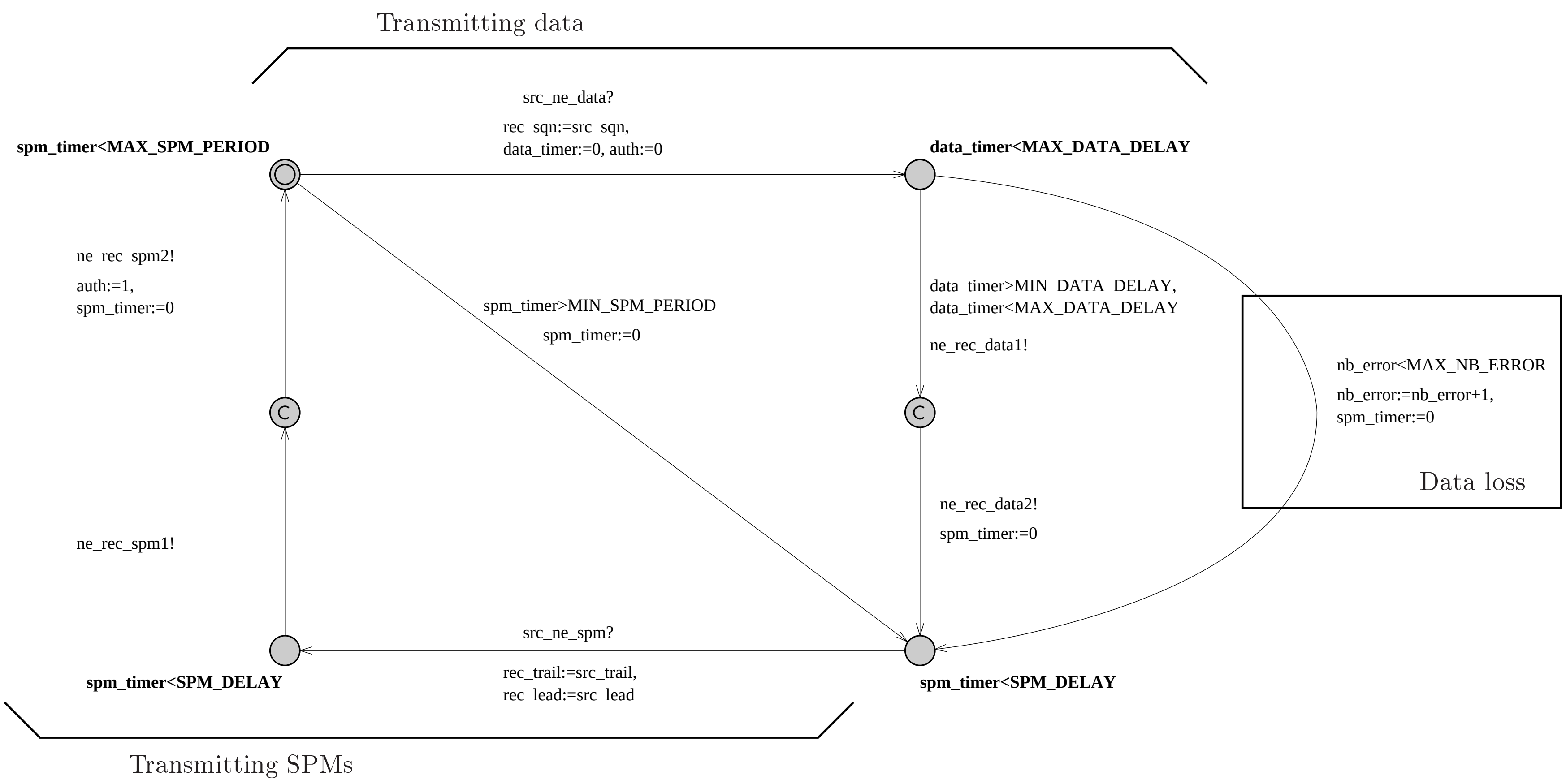


Figure 7: Model for a network element (in case of two receivers)