Technical Report

Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator

Paulo Baltarejo Sousa

Luís Lino Ferreira

HURRAY-TR-060402

Version: 5

Date: 15-04-2006

Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator

Paulo Baltarejo SOUSA, Luís Lino FERREIRA

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: {pbsousa,llf }@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

This technical report describes the Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator that implements a simulation model of the repeater-based approach. This approach defines the mechanism to extend the PROFIBUS protocol to supprot wireless communication, in which the interconnection of the wired and wireless segments is done by a intermediate system operating at Physical Layer, as repeater.

Document history:

Revision	Description	Date	By
1		15-04-2006	LLF
2		15-12-2005	LLF
3		18-06-2006	LLF
4		23-11-2006	LLF
5		26-06-2007	LLF

Index

List of Figures

List of Tables

Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator

1.Introduction

It is well accepts that the wireless communications are a surplus value for any system. Since they provide significant cuts in cabling and maintenance costs, ease in installation of equipment in hazardous areas and important add-ons in terms of flexibility and ability to evolve.

However, several constraints for the use of wireless communication at the cell level and, more acutely at the field still exist. Insufficient performance, low level of dependability, and the non existence of appropriate wireless Medium Access Control (MAC) protocols to ensure real-time behaviour are examples. Contrarily to the high performance, high dependability and real time behaviour of the (wired) fieldbuses.

Therefore, even if the addition of wireless capabilities to fieldbuses may introduce some important features to automation systems, an important concern to the system designer is that these wireless extensions do not disrupt the above mentioned characteristics of fieldbuses network.

The PROFIBUS (acronym for PROcess FIeld BUS) is one of the most popular fieldbuses with many installations in operation worldwide. However, wireless communications and wireless mobility stations are not previously considered. The Repeater-Based Hybrid Wired/Wireless PROFIBUS networks [1] covers these limitations. This approach (in this document the *Repeater-Based Hybrid Wired/Wireless PROFIBUS Network* is also referred as *approach*) extends the PROFIBUS protocol to support interoperability between wired and wireless domains.

This technical report describes the Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator (RHW2PNetSim). RHW2PNetSim implements the main functionalities of the PROFIBUS protocol and all functionalities related to Repeater-Based Wired/Wireless PROFIBUS Networks.

The structure of this document is as follows. Section 2 presents the general aspects of the PROFIBUS protocol focusing on the Data Link Layer. Section 3, describes the basic of the proposed repaeter-based approach. Section 4 describes of the OMNeT++ Framework. Then Section 5 describes the architecture and the implementation of the RHW2PNetSim.

2.Relevant Details on PROFIBUS

2.1. General Features

PROFIBUS was standardised in 1996 as an European standard [2]. It is based on the International Standards Organisation (ISO) Open System Interconnection (OSI) reference model, however collapsed to just three layers: Physical Layer (PhL), Data Link Layer (DLL) and Application Layer (AL). There is also a transversal management functionality called Fieldbus Management (FMA1/2), which is responsible for the management of the layers 1 and 2, the PhL and the DLL, respectively.

The PROFIBUS PhL can use the RS-485 standard over twisted pair or coaxial cable for the transfer of data, with bit rates up to 12 Mbit/s. For special applications, it is also possible to use other types of physical media, like optical fibre, power cable or RS-485-IS (for intrinsically safe applications).

The PROFIBUS DLL uses a token passing procedure to grant bus access to masters, and a master-slave procedure used by masters to communicate with slaves (or other masters). Slaves do not have communication initiative. They are only capable of transmitting a response (or an acknowledgement) upon master request. The token is passed between masters in ascending Medium

Access Control (MAC) address order, thus the masters organise network access in a logical ring fashion.

The PROFIBUS standard considers two different types of Application Layer profiles: PROFIBUS-FMS (Fieldbus Message Specification), which is being abandoned due to design complexity and cost, and PROFIBUS-DP (Decentralised Peripherals), which is being increasingly adopted for industrial automation and process control applications. PROFIBUS-DP is particularly suited for the cyclic exchange of data between master (Programmable Controllers, PC, etc.) and slave devices (valves, I/O devices, drives, etc.). In this dissertation DP is assumed to be used in master and slave devices.

2.2. Data Link Layer (DLL)

2.2.1. Message Cycle

In PROFIBUS, only master stations may initiate transactions, whereas slave stations do not transmit on their own initiative, but only upon (master) requests. The station that sends an *Action Frame* (the first frame transmitted in each transaction) is the *initiator* of the transaction, while the addressed one is the *responder*. A transaction (or message cycle) consists on the request or a send/request frame from the initiator (always a master station) and the associated acknowledgement or response frame from the responder (either a master station or a slave station, but typically a slave station).

All stations monitor all the requests but will only acknowledge or respond if, and only if, they are the addressees in the initiator's request. Moreover, the acknowledgement or response frame must arrive before the expiration of the `Slot Time` (T_{SL}), which is a master DLL parameter otherwise the initiator repeats the request a number of times defined by the `max_retry_limit`, another master's DLL parameter.

2.2.2. Token Passing

The token is passed between masters in ascending address order. The only exception is that in order to close the logical ring, the master with the highest address must pass the token to the master with the lowest one. Each master knows the address of the previous station (PS – `Previous Station` address), the address of the following station (NS – `Next Station` address) and, obviously, its own address (TS – `This Station` address).

If a master station receives a token addressed to itself from a station (`Source address` (SA) of the token frame, a frame format description is presented latter) registered in the `List of Active Stations` (LAS) as its predecessor (PS = SA) then this master becomes the token owner, and may start processing message cycles. On the other hand, if a master receives a token frame from a station which is not its PS, it assumes that an error has occurred, and it will not accept the token frame. However, if it receives a subsequent token from the same station, it accepts the token and assumes that the logical ring has changed. In this case, it updates the original PS value by the new one in its LAS table.

If after transmitting the token frame and within the `Slot Time`, the master detects bus activity, it assumes that its successor owns the token. Therefore, it ceases monitoring the activity on the bus. In case the master does not recognise any bus activity within the T_{SL}, it repeats the token frame and waits another T_{SL}. If it recognises bus activity within the second T_{SL}, it stops working as an active master, assuming a correct token transmission. Otherwise, it repeats the token transmission to its NS for the last time. If after the second retry there is no bus activity, the token transmitter tries to pass the token to the next successor. It continues repeating this procedure until it finds a successor from its `List of Active Stations` (LAS).

2.2.3. Token Cycle

After receiving the token, a master station is allowed to execute message cycles during `Token Holding Time` (T_{TH}) that is computed as follows:

$$T_{TH} = T_{TR} - T_{RR} \quad (1)$$

T_{TH} is equal to the difference, if positive, between the `Target Rotation Time` (T_{TR}) and the `Real Rotation Time` (T_{RR}) of the token. T_{TR} is a parameter common to all masters in the network, which must be set to the expected time for the token cycle. T_{RR} is the time measured between two consecutive token receptions – the token cycle.

PROFIBUS defines two main categories of messages: high-priority and low-priority, each using a different transmission queue that is handled differently by the DLL. At the arrival of the token, the T_{TH} timer is loaded with the value corresponding to the difference between T_{TR} and T_{RR}. If the token is delayed, then T_{TH} is set to zero and the master is only allowed to perform, at most, one high-priority message transaction. Otherwise, the master is allowed to perform high-priority message transactions until the value of the T_{TH} timer becomes negative. Low-priority messages are only transmitted when the high-priority queue is empty and T_{TH} is still positive. Note that once a message cycle is started it is always completed, including any retries, even if in meanwhile T_{TH} expires.

2.2.4. (Re)Initializing the Logical Ring

The logical ring of PROFIBUS is supported by two tables: the `Gap List` (GAPL) and the LAS. It may also optionally maintain a `Live List` (LL) table.

The GAPL consists on the address range from address TS until NS. This includes all possible addresses, except the address range between `Highest Station Address` (HSA), which cannot be a master's address, and 127, which does not belong to the Gap.

Initialization is primarily a special case of updating the LAS and the GAPL. If after power on of a master station in the LISTEN_TOKEN state a time-out is encountered, i.e., no bus activity within `Time-Out Time` (T_{TO}), it shall claim the token in the CLAIM_TOKEN state and it starts initializing the logical ring.

The master station with the lowest station address starts initialization by transmitting two token frames addressed to itself (`Destination Address` (DA) = SA = TS) it informs any other master stations (entering a NS into the LAS) that it is now the only station in the logical token ring. Then it transmits an `FDL_Request_Status` frame to each station in an incrementing address sequence, in order to register other stations. The first master station to answer with `Ready_to_Enter_Logical_Ring` is registered as NS in the LAS and thus closes the Gap range of the token holder. Then the token holder passes the token to its NS.

When a master station is in the LISTEN_TOKEN state, it shall monitor the bus activity in order to identify those master stations which are already in the logical token ring. For that purpose token frames are analyzed and the station addresses contained in them are used to generate the LAS.

After listening to two complete identical token rotations, the master must remain in the LISTEN_TOKEN state until it is addressed by an `FDL_Request_Status` transmitted by its predecessor (PS). If it succeed, it must respond with `Ready_to_Enter_Logical_Ring` and waits for the token frame addressed to it in the ACTIVE_IDLE state.

When a master station is in the LISTEN_TOKEN state all frames are neither acknowledged nor answered.

2.2.5. Ring Maintenance

The ring maintenance mechanism is distributed by all master stations. As mentioned, each PROFIBUS master maintains two tables: the GAPL and the LAS.

Each master station when holds the token frame checks its Gap addresses every time its `Gap Update Timer` (T_{GUD}) expires. If a station acknowledges positively to the GAP request (an `FDL_Request_Status` frame), with the state `Not_Ready_to_Enter_Logical_Ring` or `Slave_Station`, it is accordingly marked in the GAPL and the next address is checked. If a station answers with the state `Ready_to_Enter_Logical_Ring`, the token holder changes its GAPL and passes the token to the new NS. This (master) station, which has newly been admitted to the logical ring, has already built up its LAS when it was in the LISTEN_TOKEN state, so it is able to determine its GAPL

and its NS. This mechanism allows masters to track changes in the logical ring due to the addition (joining) and removal (leaving) of stations. This is accomplished by examining (at most) one Gap address per token visit, using an `FDL_Request_Status` frame after the execution of all high-priority transactions, and if the value of T_{TH}, is still positive.

2.2.6. Error Handling

Additionally, in order to enhance the communication system's reliability, PROFIBUS handles some operational or error states, concerning logical ring management. In [3] and [4] is presented which fault-tolerant mechanisms are activated and their effects on the network behaviour. The most important error situations within the context of this document are the token lost, "heardback removal" and error skipping:

- Token lost. This abnormal situation is clearly recovered by means of a continuous monitoring activity performed by each master in the logical ring. If a period of inactivity longer than the T_{TO} is detected, then the token is claimed by the master with the lowest address in the logical ring, and the logical ring is re-initialised.
- Heardback removal. Whenever a master is sending a token frame it must hear from the medium all transmitted bits in order to detect a defective transceiver. If the token frame sender detects differences between the transmitted and the received token frame in two consecutive transmissions then it must remove itself from the logical ring.
- Error skipping. A master station must remove itself from the logical ring when a token frame is transmitted, in which its address is "skipped" (i.e., the address of TS lies within the address range spanned by the sender and the receiver of the token frame).

2.2.7. DLL Frame Formats

PROFIBUS DLL defines three types of request/response frames, which are the `Fixed Length with no Data Field`, the `Fixed Length with Data Field` and the `Variable Data Field Length`, as illustrated in Figure 1 a), c) and d), respectively.

Each of these three types includes the following fields: `Destination Address` (DA), `Source Address` (SA) and `Frame Control` (FC). The FC field is an octet where the frame type is specified and the function code (for more details the reader is referred to Section 4.7.3 of [2]). These frames also include the `Start Delimiter` (SDx), `Frame Check Sequence` (FCS) and the `End Delimiter` (ED).

Variable data field length frames additionally contain two `Data Length` fields (LE and LEr) and they can optionally include the `Destination Address Extension` (DAE) and `Source Address Extension` (SAE), in the `Data` field. These extension fields can be used to identify the AL service which originated the frame, as well as the destination service.

PROFIBUS also defines the `Short aCknowledgement` frame (SC) and the `Token` frame, illustrated in Figure 1 b) and e), respectively. The first consists of a single byte frame, and it is used as positive acknowledgement to a request.

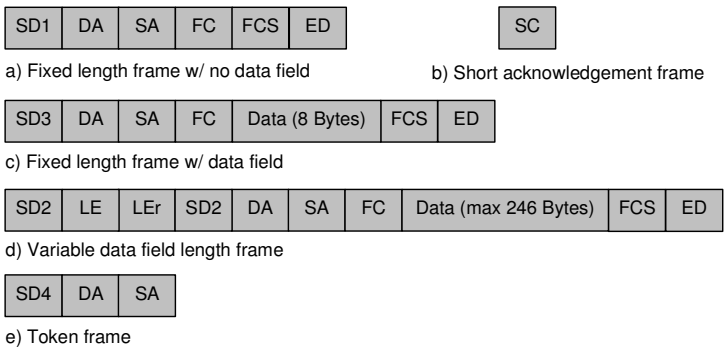

Figure 1– PROFIBUS DLL frame formats

2.2.8. Data Link Layer Services

PROFIBUS defines 4 types of data transfer services: `Send Data with Acknowledge` (SDA); `Send Data with No acknowledge` (SDN); `Send and Request Data` (SRD) and `Cyclic Send and Request Data` (CSRD).

The SDA service allows a user to transmit data to another station and receive a `Short Acknowledge` confirming its reception by the responder station. The SDN service permits to transfer data to a single station, to a group of stations (multicast) or to all stations (broadcast). The SRD service allows the transmission of a message to another station and the retrieval of a response. This service can be used, for example, to send the output settings for an I/O device and retrieve the state of the device's input ports. The CSRD builds upon the SRD service adding the capability of transferring data periodically, according to the user requirements. The CSRD service is usually not implemented in current commercial hardware platforms.

2.2.9. Timing Parameters

The PROFIBUS standard defines several timing parameters, some of which are relevant in the context of this document, such as the `Idle Time` (T_{ID}), the `Slot Time` (T_{SL}) and `Time-Out Time` (T_{TO}) parameters, which are briefly explained next.

There are two `Idle Time` (T_{ID}) parameters - T_{ID1} and T_{ID2}. T_{ID1} is a period of inactivity, inserted by a master station, after an acknowledgment, response or token frame. This parameter must be set as follows:

$$T_{ID1} = \max\{T_{SYN} + T_{SM}, \min T_{SDR}, T_{SDI}\} \quad (2)$$

where, T_{SYN} (`Synchronisation Time`) is the minimum time interval for an idle bus state before a station may accept the beginning of an action or token frame. T_{SM} (`Safety Margin Time`) is the time that elapses after the end of the T_{SYN} which is required by the receiver circuitry to be ready to start receiving a frame. $minT_{SDR}$ is the `minimum Station Delay of a Responder Time`. T_{SDI} is the `Station Delay of the Initiator Time`, after which the initiator is ready to start receiving a frame from the responder. Figure 2 depicts an example where the `Transmission Delay Time` (T_{TD}) due to the network propagation delay is also illustrated.

T_{ID2} is the idle time inserted by a master station after transmitting an unacknowledged request frame. T_{ID2} must be set as follows:

$$T_{ID2} = \max\{T_{SYN} + T_{SM}, \max T_{SDR}\} \quad (3)$$

where, $maxT_{SDR}$ is the `maximum Station Delay of a Responder Time`.

The `Slot Time` (T_{SL}) is used by a master station to detect if a transaction with a slave (or with its successor, in the token passing) has failed. A timer is loaded with T_{SL} at the end of the transmission of a request frame. Upon its expiration, the master station may execute another retry for the same request, if the number of retries executed is smaller than the `max_retry_limit` parameter, or it may inform the upper layers of a transmission failure. A timer is also loaded with T_{SL} after transmitting the token frame. If it expires before a master has detected any activity in the bus then it signals the MAC layer in order to take the appropriate actions according to the token passing procedure.

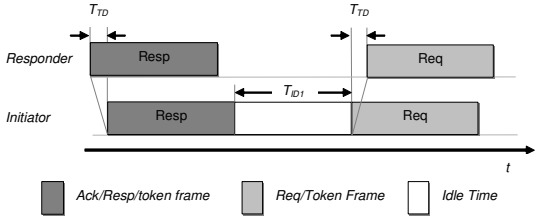

Figure 2 – `Idle Time` parameter – T_{ID1}

The `Slot Time` parameter (T_{SL}) must be set to the maximum between two values – T_{SL1} and T_{SL2}. T_{SL1} can be calculated as follows:

$$T_{SL1} = 2 \times T_{TD} + \max T_{SDR} + 11bit + T_{SM} \quad (4)$$

where *bit* is the time duration of a bit. T_{SL2} can be calculated as follows:

$$T_{SL2} = 2 \times T_{TD} + \max T_{ID1} + 11bit + T_{SM} \quad (5)$$

Note that all masters in the network must hold the same T_{SL} value, due to the token passing mechanism.

The `Time-Out Time` (T_{TO}) controls the token passage, in PROFIBUS, a token lost is detected when a master does not detect any network activity for a time period defined by its T_{TO}, which is set to as follows:

$$T_{TO} = 6 * T_{SL} + 2 * n * T_{SL} \quad (6)$$

where n is the master address. In Eq. 6, the first term makes sure that there is sufficient difference to the maximum possible T_{ID} between two frames (recall Eq. 4 and Eq. 5). The second term ensures that the master stations start their token claiming procedure in different moments after an error has occurred.

2.3. Application Layer (AL): PROFIBUS-DP

The PROFIBUS-DP (DP for short) protocol [5] is specially suited for the exchange of data between controllers (typically masters) and field devices like I/O, drives or valves (typically slaves). DP provides the functionalities to configure field devices and to perform cyclic exchange of data between the controller and the field devices.

The main functionalities of PROFIBUS-DP are related to the reading and writing of variables from/to slave devices. The retrieval of data is made cyclically by the DP protocol, according to the timing parameters configured by the user.

3. Relevant Details on the Repeater-Based Hybrid Wired/Wireless PROFIBUS Architecture

A hybrid wired/wireless fieldbus network is composed of wired and wireless stations. A wired domain is a set of stations physically connected through a wired bus. A wireless domain is composed by a set of wireless stations that intercommunicate either directly or indirectly via wireless (e.g., radio) channels. Wireless communications may be achieved by two ways: in a direct way or via Base Station (BS). If wireless stations are able to intercommunicate directly, the wireless domain is called an *ad-hoc* domain. Otherwise, if messages are relayed by a BS the wireless domain is usually called a *structured* domain. In this dissertation we always assume that we are using the structured approach since this is the only one that permits the mobility of wireless stations.

A BS operates as a wireless repeater using two radio channels, one to receive frames from the wireless stations (the uplink channel), and another to transmit frames to wireless stations (the downlink channel). The interconnection between wireless and wired domains is done through a special device designed as Intermediate Systems (IS). This device has to be provided with two communication interfaces: one to connect to the wired domain (wired communication interface) and another to connect to the wireless domain (wireless communication interface).

In the Repeater-Based Hybrid Wired/Wireless PROFIBUS approach [1], the ISs operate essentially as repeaters, that is, they receive frames from one communication interface and retransmit those frames using the other communication interface.

Figure 3 depicts a wired/wireless fieldbus network scenario. The ISs (operating as repeaters) may include the BS functionalities in their wireless communication interfaces. This network scenario comprises four domains, two wired domains (D^2 and D^4) and two structured wireless domains (D^1 and D^3). Three repeaters (R1, R2 and R3) interconnect the domains. The wireless communications are relayed by two BSs (BS1 and BS2), included in the repeaters. The network also comprises three wired masters (M1, M2 and MM), two wireless mobile masters (M3 and M4), five wired slaves (S1, S2, S3, S4 and S5) and one wireless mobile slave (S6).

In order to guarantee the operation and interoperability of the hybrid wired/wireless fieldbus network there can be no more than one possible path between any two domains (tree-like topology), i.e., no closed loops can exist. The mobility procedure only makes sense when there are more than one wireless domain and when these domains are structured. The mobility procedure will be detailed later.

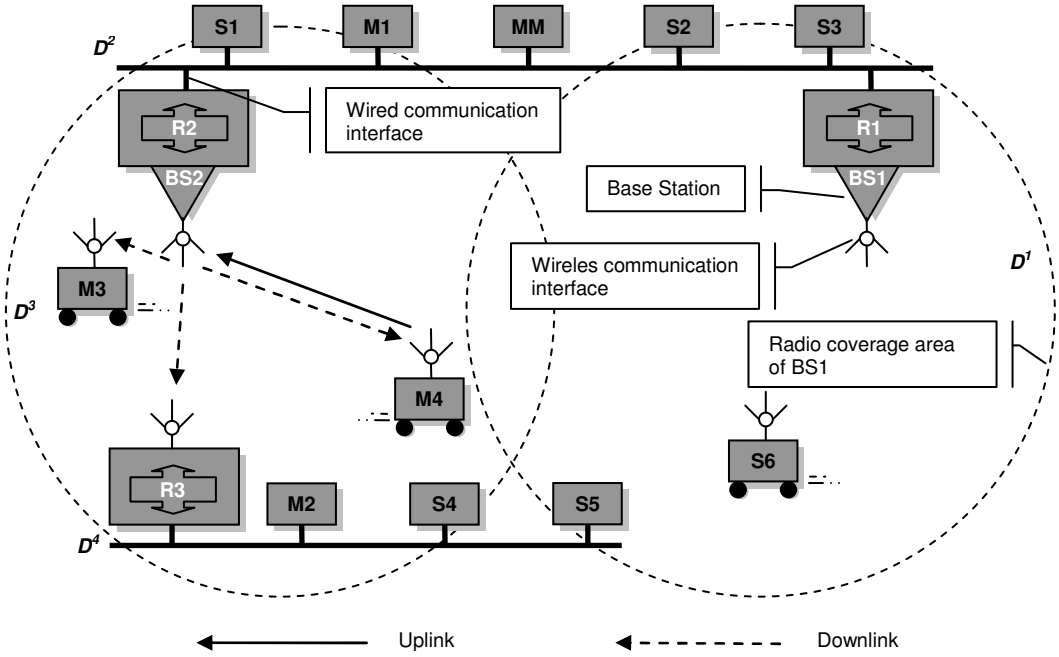

Figure 3 – Repeater-based hybrid wired/wireless PROFIBUS network example

3.1.1. Repeater Operation

As mentioned, in this approach the interconnection between domains is done by ISs operating as repeaters. Traditionally operating just as a signal regenerator, a repeater can also interconnect two networks with different PhL protocols (e.g., different bit rates).

A repeater is classified according to its relaying behaviour: *store-and-forward*, when a PhL frame must be completely received from one port before being transmitted to the other port; *cut-through*, when a repeater starts relaying a PhL frame which has not been completely received yet.

A repeater does not perform any address filtering. This result in a broadcast network, i.e., every station listens to every frame transmitted by any other station in the network. The use of repeaters implies a single MAC address space and that only one logical ring exists in the network. For that the network operation is based on the Single Logical Ring (SLR).

3.1.2. Wired/Wireless Domains Interconnection

A repeater may need to implement more than a bit-by-bit repeating functionality. This is the case when it interconnects communication media with different PhL frame formats. In order to encompass the functionalities referred, each repeater has an associated internal `relaying delay time` (t_{rd}). It is

assumed that the repeaters always introduce a minimum inactivity period – `minimum idle time` (T_{IDm}) – between any consecutive PhL frames.

When a repeater receives a PhL frame from one port it must start the transmission in an instant, the `start relaying` instant ($t^{i\to j}_{sr}$), that guarantees that the retransmission is done without time gaps. $t^{i\to j}_{sr}$ is defined as the earliest time instant for start relaying a specific PhL frame from domain D^i to domain D^j, counted since the beginning of the reception of the PhL frame in domain D^i. The $t^{i\to j}_{sr}$ instant for a specific repeater depends on its operation mode – either store-and-forward or cut-through. For a cut-through repeater, the following is assumed:

- When relaying a frame from D^i to D^j, it cannot start being relayed while the first char of the DLL frame of D^i is not completely received by the repeater;
- The PhL frame cannot start being relayed while the length of the DLL frame is not known (by the repeater);
- When relaying a frame from D^i to D^j, the instant for start relaying the PhL frame must take into account that the repeater cannot run out of bits to relay from D^i to D^j, i.e., the transmission of a PhL frame in D^j must be continuous, without time gaps.

Taking these assumptions into account, which are illustrated in Figure 4, the `start relaying` instant for a cut-through operation mode repeater is defined as:

$$t^{i\to j}_{sr} = \max\left\{t^{i}_{dr}, t^{i}_{lk}, t^{i\to j}_{ng}\right\} \quad (1)$$

where:

- t^{i}_{dr}, the `data ready` instant, is the instant at which a predefined amount of DLL data has been received from D^i (ready to be relayed), counted since the beginning of the PhL frame in D^i. For the cut-through behaviour, it is considered that it is the instant at which the first DLL character is completely received;
- t^{i}_{lk}, the `length known` instant, is the instant at which the length of the DLL frame in D^i is known, counted since the beginning of the PhL frame in D^i;
- $t^{i\to j}_{ng}$, the `no gaps` instant, is the earliest instant to start relaying the PhL frame from D^i to D^j in a way that guarantees that the transmission in D^j is continuous.

Consider the example depicted in Figure 2.4. The first time instant is the data ready (t^{i}_{dr}), followed by the time instant when the length of the frame is known (t^{i}_{lk}). The last instant (thus the maximum of the three) is the time instant that guarantees a continuous retransmission of the PhL frame ($t^{i\to j}_{ng}$). This situation usually happens when the duration of the PhL frame in D^j is smaller than in D^i.

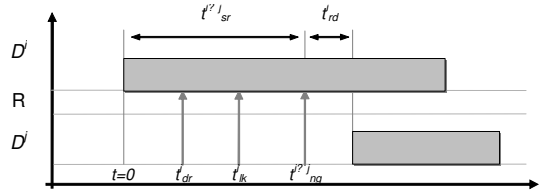

Figure 4 – Timing behaviour of a repeater

3.1.3. Traffic Adaptation

Network interconnection often brings up the problem of network congestion. Generally, if for any time interval, the total sum of demands on a resource is more than its available capacity, the resource is said to be congested for that interval. In the case of computer networks, resources include buffer space and processing capacity in the ISs and for example, if during a short interval, the buffer space of an IS is smaller than the one required for the arriving traffic, frame loss may occur (dropped frames) and the IS is said to be congested.

It is also true that the congestion problems depend dramatically on the type of IS used in the interconnection. Particularly if the ISs act as repeater, traffic congestion may occur as a result of the

heterogeneous characteristics of the interconnected physical media. The heterogeneity in bit rates and in PhL frame formats in a broadcast network imposes the consideration of some kind of traffic adaptation scheme.

The timing diagram depicted in Figure 5 illustrates a sequence of transactions where one repeater interconnects the two domains and it is assumed that the PhL frame duration in D^j is twice the PhL frame duration in D^i and that $t^{i \to j}_{sr}$ is constant (for the sake of simplicity). Note that since the idle time is defined as the duration of a predefined number of (idle) bits separating consecutive frames in the network, its duration is assumed to be different for the two domains.

Figure 2.5 illustrates an increasing queuing delay ($q_1 < q_2 < q_3$), caused by the different physical media, that will impact on the `system turnaround time` (t_{st}) for certain transactions. The t_{st} for a message transaction is the time elapsed since an initiator ends transmitting a request frame until it starts receiving the correspondent response frame. For example, in the case of the request that corresponds to transaction 3, which is addressed to a responder in domain D^j, the system turnaround time for this transaction (t_{st3}) will be affected by the cumulative queuing delay (q_3) in the repeater.

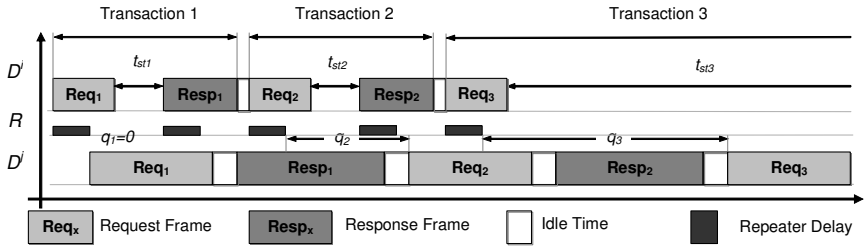

Figure 5 – Increasing queuing delay by a repeater

Traffic congestion in a repeater can be avoided through the insertion of additional inactivity (idle) intervals before issuing a message transactions [6]. Obviously, the insertion of this additional idle time reduces the number of transactions per time unit when the responder is not in the same domain as the initiator. The PROFIBUS MAC mechanism allows only one station (master or slave) to transmit at a given moment in time.

Every master in PROFIBUS has two different `Idle Time` parameters – T_{ID1} and T_{ID2}. As mentioned in Section 2.2.9, a master always waits T_{ID1} after receiving a response/acknowledgement or a token frame, before transmitting another frame. It must also wait T_{ID2} after transmitting an unacknowledged request frame, and before transmitting another frame (request or token). For a traditional wired network all masters may set their T_{ID1} and T_{ID2} parameters to the minimum default value, which is usually adequate to cope with bit synchronisation requirements.

In this approach, the traffic adaptation is based on the computation of the additional idle time that must be inserted by each master, in order to properly encompass the interconnection of heterogeneous physical media. The timing diagram depicted in Figure 6 illustrates a sequence of transactions where queuing delay is zero for all transactions, on the first repeater. This is due to the additional extra idle time.

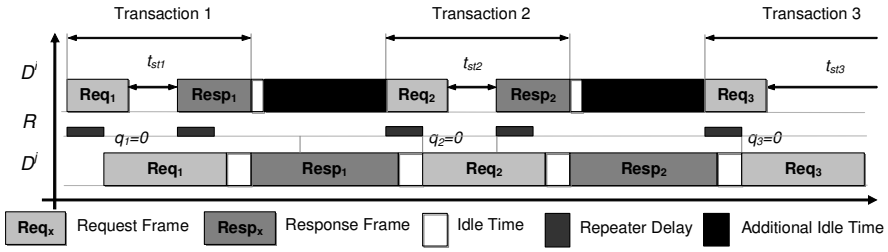

Figure 6 – Using additional idle time for media adaptation

Another consequence of using ISs that act as repeaters is related to the master PROFIBUS DLL parameter, the `Slot Time` (T_{SL}). Within the context of this approach, T_{SL} assumes a particular importance. On one hand, T_{SL} must be set large enough to cope with the extra latencies introduced by the repeaters. On the other hand, T_{SL} must be set as small as possible such as the system responsiveness to failures does not decrease dramatically, that is, a master must detect a message/token loss or a station failure within a reasonably small time. The timing diagram depicted in Figure 7 illustrates a transaction sequence that is relayed by two repeaters. One interconnects domains D^i and D^j and another interconnects domains D^j and D^k. It is assumed that the PhL frame duration in D^i and D^k is half the PhL frame duration in D^j and that $t^{x\rightarrow y}_{sr}$ is constant (for the sake of simplicity).

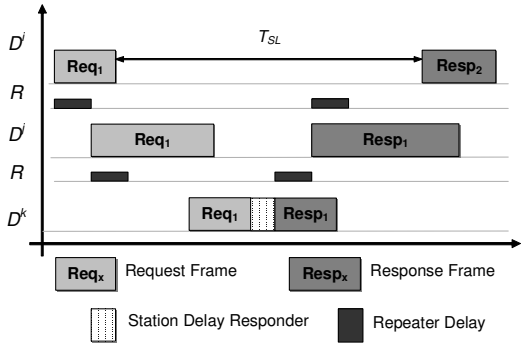

Figure 7 – `Slot Time` (T_{SL})

The setting of these parameters must be performed according to the procedures described in [1].

3.1.4. The Mobility Procedure

The mobility procedure [7] provides a seamless handoff for all kinds of wireless mobile stations (master/slave).

Due to the broadcast nature of the network, the mobility procedure just encompasses a mechanism for radio channel assessment and switching. The basics of this procedure are outlined next.

The `Mobility Master` (MM) (i.e., the master that has the responsibility of triggering the mobility procedure) sends periodically a special (unacknowledged) frame – the `Beacon Trigger` (BT). This BT frame is broadcast to the entire network and causes each BS to start transmitting a pre-defined number of `Beacon` frames in its downlink radio channel. Wireless mobile stations receive these `Beacon` frames, assess the signal quality of all (downlink) radio channels and switch to the radio channel set with best quality. Figure 8 shows the simplified operation of the mobility mechanism considering the network scenario depicted in Figure 3, in which master MM is operating as `Mobility Master`.

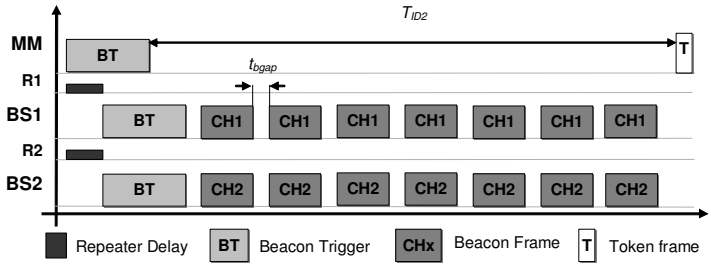

Figure 8 – Mobility procedure

In [1] the author shows how to calculate the number of beacons to be transmitted in each domain which guarantees that every wireless station is capable of evaluating the signal quality of each radio channel set. The setting of this parameter is different between each repeater and it must be set prior to runtime as well as the mobility period (i.e., the time interval between the queuing of each BT by the MM).

4.Simulation Software

4.1. OMNeT++ (Objective Modular Network Testbed in C++)

OMNeT++ [8] is an object oriented modular discrete event simulator, which provides a reusable component framework, where the system components can be independently built, characterized and assembled into larger components and models. The basic system components are built using the C++ programming language and then assembled into larger components and models using a high level language, named NED (an OMNeT++ specific scripting language).

An OMNeT++ model consists of hierarchically nested *modules* which communicate between them using messages. OMNeT++ models are often referred to as networks. The top level module is the *system* module. The system module contains *sub-modules*, which can also contain sub-modules themselves. The depth of module nesting is not limited, consequently providing a useful way to reflect the logical structure of the system in the model structure (Figure 9).

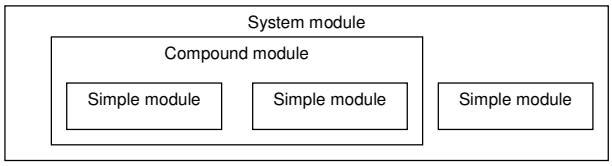

Figure 9 – Simple and compound modules

Modules that contain sub-modules are termed *compound* modules, in opposite to *simple* modules which are at the lowest level of the module hierarchy. The simple modules of a model contain the algorithms coded using C++ programming language. The full flexibility and power of the C++ programming language can be used, in conjunction with the OMNeT++ simulation class library. Further, OMNeT++ has a consistent object-oriented design. Thus, Object-Oriented Programming concepts (like inheritance and polymorphism) can be used to extend the basic functionality of the simulator.

4.1.1. Messages, Gates and Links

Modules communicate by exchanging messages which represent frames or packets in a computer network. These messages can contain arbitrarily complex data structures. Simple modules send

messages through *gates* or directly based on their unique identifier. Messages can arrive from another module or from the same module (self-messages are used to implement timers).

Gates are the input and output interfaces of modules. Messages are sent out through output gates and arrive through input gates. Each connection (also called a link) is created within a single level of the module hierarchy and is composed by two gates. Within a compound module, one can connect the corresponding gates of two sub-modules, or a gate of one sub-module and a gate of the compound module (Figure 10).

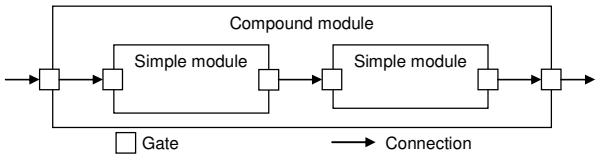

Figure 10 – Module's gates and connections

Due to the hierarchical structure of the model, messages typically travel through a series of connections, but these messages are sent and received by simple modules. Such series of connections, which go from simple module to simple module, are called routes. Compound modules act as "cardboard boxes" in the model, transparently relaying messages between their inside and their outside world, i.e., they are used to aggregate other modules relaying messages from inside to the outside and vice-versa without any processing.

4.1.2. Modelling Delays, Bit Error Rate and Data Rate

Connections can be assigned three parameters which facilitate the modelling of communication networks: *propagation delay* (sec), *bit error rate* (errors/bit) and *data rate* (bits/sec). Each of these parameters is optional. One can specify link parameters individually for each connection, or define link types (also called channel types) once and use them throughout the whole model.

The propagation delay is the amount of time the arrival of a message is delayed when it travels through a communication channel. The bit error rate has influence on the transmission of messages through the channel. The bit error rate is the probability that a bit is incorrectly transmitted. The data rate is specified in bits/second, and it is used for transmission delay calculation. The sending time of a message normally corresponds to the transmission of its first bit, and the arrival time of the message corresponds to the reception of the last bit (Figure 11).

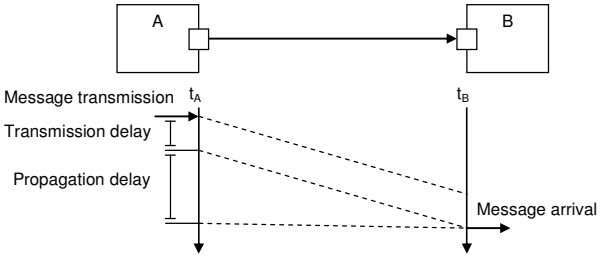

Figure 11 – Message transmission

In OMNeT++, the length of a message does not depend of its data structure composition, but on the length attribute. This attribute is used to compute the transmission delay when the message travels through a connection with an assigned data rate.

4.1.3. An OMNeT++ Example Model

An OMNeT++ model consists of the following parts:

- NED language topology description(s) which describes the module structure and respective parameters, gates, etc. These are files with .ned suffix. NED files can be written with any text editor or using the GNED graphical editor.
- Simple modules are C++ sources files, with .h/.cc/.cpp suffix.

The simulation system provides the following components:

- Simulation kernel, which contains the code that manages the simulation and the simulation class library. It is written in C++, compiled and put together to form a library.
- User interfaces. OMNeT++ user interfaces are used with simulation execution, to facilitate debugging, demonstration, or batch execution of simulations. There are several user interfaces, written in C++.
- Simulation programs are built from the above components. First, the NED files are compiled into C++ source code, using the NEDC compiler which is part of OMNeT++. Then all C++ sources are compiled and linked with the simulation kernel and a user interface to form a simulation executable.

The simulation executable is a standalone program, which can be run in any machine. When the program is started, it reads from a configuration file (usually *omnetpp.ini*) settings that control how the simulation is run and values for model parameters. The configuration file can also specify several simulation runs; in the simplest case, they will be executed by the simulation program one after another or executed on a parallel environment.

4.1.4. Event-Based Simulation

As mentioned, an OMNeT++ model consists of hierarchically nested modules which communicate between them using messages. Each message can be exchanged directly between simple modules or via a series of gates and connections. The local simulation time advances when a module receives messages from another module or from itself. Self-messages are used by a module to schedule events at a later time.

In the initialization step, OMNeT++ builds the network: it creates the necessary simple and compound modules and connects them according to the NED definitions. OMNeT++ calls the `initialize()` functions of all simple modules, which is usually used to initialize the data members. The `handleMessage()` function is called during event processing. This means that the behaviour of each module is coded in this function. The `finish()` function is called when the simulation terminates successfully, it is usually used to write statistics at the end of a simulation run.

In order to clarify these concepts, Figure 12 presents a typical PROFIBUS network transaction, which consists on the request frame from the initiator (master M1) and the associated response frame by the responder (slave S1). The initiator has to wait an `Idle Time` (T_{ID}) before sending a request frame and the responder has to wait `Station Delay Responder Time` (T_{SDR}) before sending a response frame.

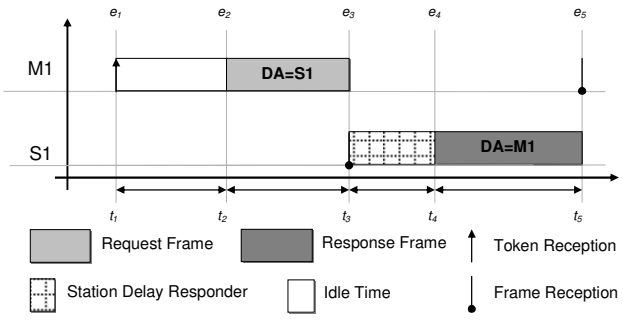

Figure 12 – PROFIBUS transactions events

Assuming that master M1 (initiator) has just received the token (event e_0 at the instant t_0), then it will schedule a self message for instant t_1, which marks the beginning of a request frame transmission and the end of the T_{ID} (event e_1).

Event e_2, at instant t_2 represents the reception of the request frame's last bit by slave S1. This instant is calculated as function of the request frame's length and the data rate of the connection. As soon as the slave S1 receives the request frame's last bit it schedules a self message to simulate the end of the T_{SDR} and begin of the response frame transmission (event e_3 at the instant t_3). Event e_4 corresponds to the reception of the response frame's last bit by master M1.

In order to model the system described above two simple modules can be used, one to model the master station (hereafter called `MasterStation` module) and another to model the slave station (hereafter called `SlaveStation` module).

Figure 13 shows part of the `handleMessage(cMessage *msg)` function (or method) of the `MasterStation` module. This function is automatically invoked at reception of every message. Therefore, the arriving of the token frame to the `MasterStation` module instance called M1 is handled by this function. This message is a PROFIBUS message where the DA and SA are equal to TS and PS (line 20), respectively. M1 computes the T_{TH} (line 22) and in order to wait T_{ID} before sending a frame it schedules a self message to arrive at instant T_{ID} counted from the current instant (the current simulation time is given by `simTime()` function) (line 24). When M1 receives a self message (line 6) that marks the end of the inactivity time (T_{ID}), it removes a message from its message output queue (line 10) and starts transmitting the request frame addressed to the `SlaveStation` module instance called S1(line 11).

```
1.  void MasterStation::handleMessage(cMessage *msg)
2.  {
3.        cMSG_Profibus *msg_profibus=NULL;
4.        cMSG_Self *msg_self=NULL;
5.        // handle the message according to the simple module algorithm
6.        if (msg->isSelfMessage()) {   //usually used as timer
7.                  msg_self=(cMSG_Self s *) msg;
8.                  switch(msg_self->getAction()){
9.                            case SEND_MESSAGE:
10.                                     dequeueMessage(&msg_profibus);
11.                                     send(msg_profibus, "out");
12.                           }
13.                           ...
14.                 }
15.       }
16.       else{
17.                 msg_profibus=(cMSG_Profibus *) msg;
18.                 switch(msg_profibus->getType()){
19.                           case TOKEN_FRAME: //if it is a token frame
20.                                     if(msg_profibus->getDA()==TS &&msg_profibus->getSA()==PS){
21.                                               ...
22.                                               computeTHT();
23.                                                msg_self->setAction(SEND_MESSAGE);
24.                                               scheduleAt(simTime()+TID, msg_self);
25.                                     }
26.                           break;
27.                           case RESPONSE_FRAME: //if it is a response frame
28.                                     if(msg_profibus->getDA()==TS && ...){
29.                                               scheduleAt(simTime()+TID, msg_self);
30.                                     }
31.                           break;
32.                           ...
33.
34.                 }
35.       }
36. }
```

Figure 13 – `handleMessage(cMessage *msg)` function, C++ code (`MasterStation`)

Figure 14 presents `handleMessage(cMessage *msg)` function of the `SlaveStation` module. In same way this function is automatically invoked as soon as a `SlaveStation` module instance receives a message. At reception of the message, slave S1 checks if the received frame is addressed to it (line 20). It schedules the sending of the response (line 23) if it is. Otherwise, no action is taken.

```
1.   void SlaveStation::handleMessage(cMessage *msg)
2.  {
3.  cMSG_Profibus *msg_profibus=NULL;
4.  cMSG_Self *msg_self=NULL;
5.  // handle the message according to the simple module algorithm
6.  if (msg->isSelfMessage()) {         //usually used as timer
7.        msg_self=(cMSG_Self s *) msg;
8.        switch(msg_profibus->getAction()){
9.                  case REPLY_MESSAGE:
10.                           buildResponseMessage(&msg_profibus);
11.                           send(msg_profibus, "out");
12.                 }
13.                 ...
14.       }
15.    }
16. else{
17.       msg_profibus=(cMSG_Profibus *) msg;
18.       switch(msg_self->getType()){
19.                 case REQUEST_FRAME: //if it is a request frame
20.                           if(msg_profibus->getDA()==TS && ...){
21.
22.                                                   msg_self->setAction(REPLY_MESSAGE);
23.                                                   scheduleAt(simTime()+TSDR, msg_self);
24.                                         }
25.                                         ....
26.                           }
27.                 break;
28.                 ....
29.
30.       }
31.    }
32. }
```

Figure 14 – `handleMessage(cMessage *msg)` function, C++ code (`SlaveStation`)

5.Repeater-Based Hybrid Wired/Wireless PROFIBUS Architecture Simulation Model

5.1. Basic Architecture

The `HW2PNet` module represents the entire network, that is, the system module in the context of the OMNeT++ framework. The `Controller` is the module that coordinates the simulation and performs several tasks, such as, parameterization, configuration of the other module instances and it is also responsible for the setup of the simulation run. The `Domain` module models a network domain and interconnects all components in a single network domain. The `Master` and `Slave` modules model a master or slave standard PROFIBUS network device.

On the left side, Figure 15 shows how the main modules are interconnected. There are 2 kinds of the connections: `ctrl_con` and `domain_con` connections. The `ctrl_con` connections are used to establish the connections between the `Controller` module instance and all module instances in the overall system. This kind of connection has no delay and is used for simulation control and configuration purposes. The `domain_con` connections are used to establish the connections among all domain components (between `Master` and `Slave` module instances and the `Domain` module instance).

On the right side Figure 15 there is a connection example between a `Master` and a `Domain` module instances. The `Master` instance is called M1 and `Domain` instance D1. Each of these connections is composed of four gates, two for each connected module instance. One gate is for input and it is connected to the output gate of the other module instance and vice-versa.

The messages are sent to D1 from M1 through a gate called `domain_gateOut`. Consequently D1 receives messages from M1 through a gate called `station_gateInM1`. D1 sends messages to M1 through a gate called `station_gateOutM1`.

Some station parameters, like T_{SDR} or T_{ID} are modelled either by Probability Distribution Function (PDF) or by a constant value. The PDFs implemented require at most four parameters. One of them defines which PDF is used and the other three are the arguments of the PDF. The name of all these parameters uses the `_pdf` prefix. For example, the parameters associated to T_{SDR} are the

following: `_pdf_tsdr_type`, `_pdf_tsdr_par1`, `_pdf_tsdr_par2` and `_pdf_tsdr_par3`. The `_pdf_tsdr_type` indicates which PDF will be used to generate the value of the T_{SDR} and the other parameters are the arguments of the PDF. The PDFs supported by both simulators are described in detail in [9]. Additionally the same parameters can also be used to make a configuration using constant values.

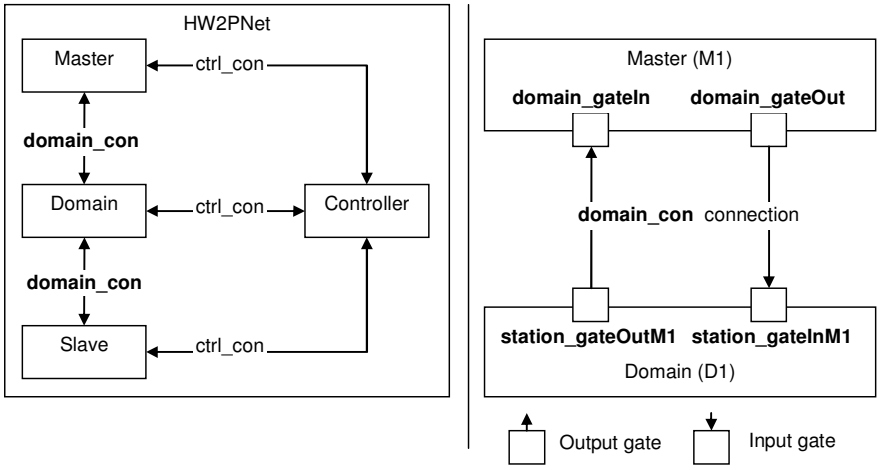

Figure 15 – Modules, connections and associated gates

One of the most important steps of a simulation study is to analyze output data generated by the simulator [10]. The RHW2PNetSim enables gathering information about the response time of transactions, information about state machine transitions of each module, information about the PDFs in use and information about the Bit Error Model (BEM) in use. The name of the parameters related to these functions use the `_output` prefix.

`Master`, `Slave` and `Domain` modules are all identified by a parameter called `_name`. The value of this parameter must be unique in the overall network, since it identifies a module instance.

To simplify the parameterization of the module instances, all common parameters to the network are associated with the `Controller` module and all common parameters to the domain are associated to the `Domain` module. These parameters are used by the `Controller` module instance to do the station parameterization. This characteristic makes the simulation configuration less complex and less error prone.

In order to model a repeater, two simple modules were developed the `Connection_Point` and `ComFunc` modules, Figure 16 illustrates these modules.

The `Connection_Point` is a simple module that establishes the connection with the `Domain` module. A repeater must include at least two of these module instances. The `ComFunc` is also a simple module that links `Connection_Point` module instances of the same repeater.

In addition to the `ctrl_con` and `domain_con` connections, there is another kind of connection: the `repeater_con`. This kind of connection is used to connect a repeater to a `Domain` module instance. For that purpose the `Domain` module is provided with a set of input and output gates, whose names use the `repeater_gate` prefix.

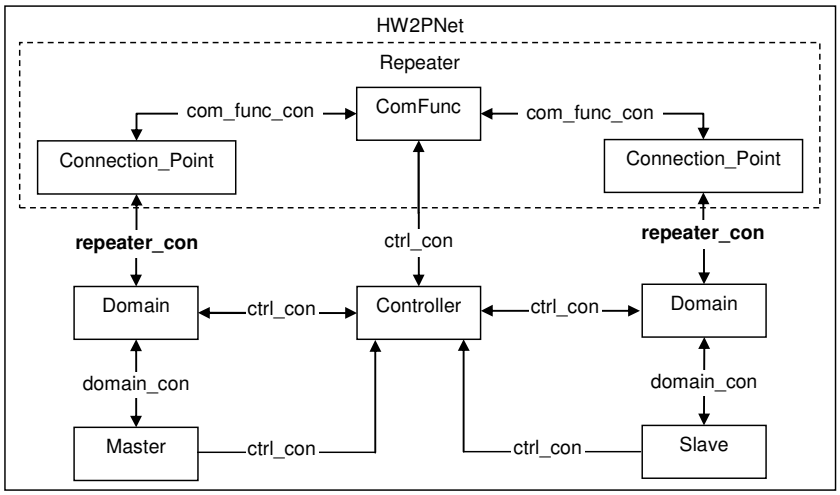

Figure 16 – Modules and connections of the RHW2PNetSim

In the next sections, further details are provided concerning model architecture and implementation.

5.1.1. HW2PNet

In OMNeT++ to actually get a simulation that can be run, it is necessary to write a network definition. A network definition declares a simulation model as an instance of the system module, in this case of the `HW2PNet` module. A network definition is declared with keyword *network*, followed by the network instance's name and ends with the keyword *endnetwork*. Figure 17 presents the network definition in which the system module instance is called `theRHW2PNet`. No simulation parameters are assigned in the network definition, since they are assigned by the configuration file named *omnetpp.ini*.

```
network
        theRHW2PNet:    HW2PNet
Endnetwork
```

Figure 17 – Network definition

`HW2PNet` module is a compound module that contains all other module instances. An OMNeT++ network has at least one instance of each module. The number of module instances is specified in the `HW2PNet` module instance.

Figure 18 presents the OMNet++ NED definition of the `HW2PNet` module. This kind of compound module definition must be contained between the keywords *module* and *endmodule*. It is composed by the module parameters and by its sub-modules. Additionally, in the declaration of the compound modules elements, like gates and connections can be specified.

Parameters are mainly used to define the module behaviours. These parameters can be strings or numeric values as well as random values from different PDFs. Within a compound module, parameters can define the number of sub-modules as well as the number of gates. In this case, the gates and connections are assigned dynamically at run time.

```
module HW2PNet
        parameters:
                _num_domains:       numeric,
                _num_masters:       numeric,
                _num_slaves:        numeric,

                _num_repeaters:     numeric,

        Submodules:
                master:             Master[_num_masters];
                slave:              Slave[_num_slaves];
                domain:             Domain[_num_domains];
                repeater:           ComFunc[__num_repeaters];
                cp:                 Connection_Point[_num_repeaters *2]
                controller:         Controller;
endmodule
```

Figure 18 – `HW2PNet` module NED definition

Figure 19 depicts parts of the configuration file with the settings related to the `HW2PNet` module instance, which is the simulation network. This repeater-based network is composed by only four domains (`_num_domains` parameter), five masters (`_num_masters` parameter) and six slaves (`_num_slaves` parameter) and three repeaters (`_num_repeaters` parameter).

```
theRHW2PNet._num_domains=4
theRHW2PNet._num_masters=5
theRHW2PNet._num_slaves=6
theRHW2PNet._num_repeaters=3
```

Figure 19 – Configuration file related with `HW2PNet` module instance (excerpt)

Figure 20 depicts a graphical representation of the network presented in Figure 2.3. In this figure it is clear that the `Controller` instance (labeled *controller*) is able to communicate with all module instances, for parameterization and control purposes. `Master` and `Slave` module instances are connected to their correspondent `Domain` module instances, symbolized by a rectangle or a cloud for the case of wired or wireless domains, respectively. Each repeater is composed by two `Connection_Point` module instances (labeled *cp[x]*, where *x* is a number between 0 and 5) and one `ComFunc` module instance (labeled *repeater[x]*, where *x* can be 0, 1 or 2). Note that, the `Connection_Point` module instance, which is symbolized by a large ball, also models the BS functions.

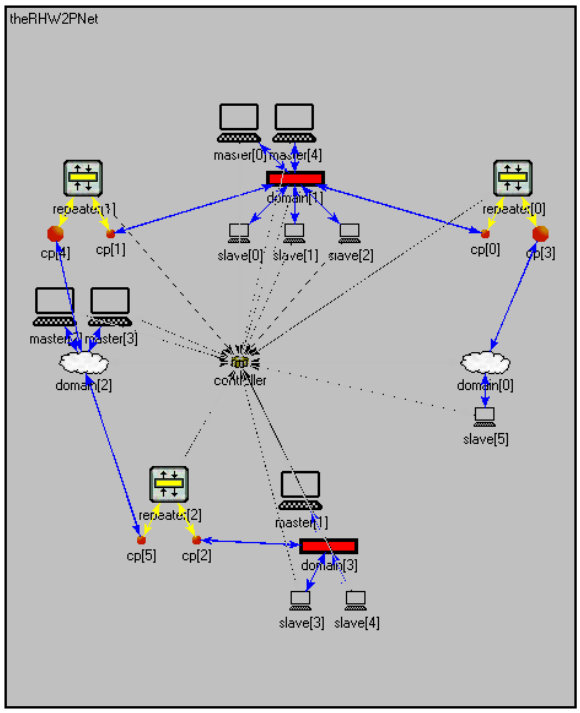

Figure 20 – Screenshot of the output window of the RHW2PNetSim

5.1.2. Controller

The `Controller` is a simple module that coordinates the simulation and performs several managing tasks, acting as the simulation supervisor. Parameters that are specific of one module instance or common to all module instances in the network are assigned to the `Controller` module instance. On simulation setup, the `Controller` module instance makes the parameter setting of the all other module instances. Additionally, due to memory limitations, the `Controller` module instance is responsible for periodically sending commands to other module instances, commanding them to dump the information gathered to data files. Finally, whenever a `Master` or `Slave` module instance changes between domains, this module updates the network configuration and the corresponding connections. Note that, OMNeT++ does not provide any native mechanism for mobility.

A simple module is declared with keywords *simple*, followed by the modules' name, and *endsimple*. The parameters and the gates can be specified in the declaration of a simple module.

Figure 21 presents the NED definition of the `Controller` simple module. Parameter `_domain` is a string which defines the configuration of the domain. It is written using predefined structure based in tags. The `Controller` module instance extracts information from these strings to perform the network configuration.

```
simple Controller
        parameters:
                _output_gant_diagram:   numeric,
                _output_resp_time:      numeric,
                _output_states:         numeric,
                _output_pdf:            numeric,
                _output_bem:            numeric,
                _output_period:         numeric,
                _output_path:           string,
                _tmob:                  numeric,
                _mm:                    string,
                _domain:                string,
                _inter_domain:          String;
endsimple
```

Figure 21 – `Controller` module NED definition

The parameters with the `_output` prefix are related to the output data files. If one of these parameters has a value of one it means that the information referred to, by the parameter, must be gathered by the module instances (Table 1). A detailed description of the output data files is presented in [11].

Table 1 – Summary of the output data information

Parameter	Information
`_output_gant_diagram`	Information necessary to build event Gant Diagrams.
`_output_resp_time`	Information about the response time of each transaction.
`_output_states`	Information about module instances' state machine and their transitions.
`_output_pdf`	Information about the probability distribution functions used.
`_output_bem`	Information about the bit error model used.
`_output_period`	Period for dumping the gathered information on to data files.
`_output_path`	The path of the directory output files

Figure 22 presents the parameter values related to the `Controller` module instance, assuming the network depicted in Figure 3. This file also includes information about MM (`Master` module instance named M5) and about the periodicity (`_tmob`).

```
theRHW2PNet.controller._mm="M5"
theRHW2PNet.controller._tmob=200ms
…
theRHW2PNet.controller._domain="\
<d><n>D1</n><m></m><s>S6</s><cp>CP8</cp><bs>CP8</bs><pos>400:300</pos></d>\
<d><n>D2</n><m>M1:M5</m><s>S1:S2:S3</s><cp>CP6:CP5</cp><bs></bs><pos>200:150</pos></d>\
<d><n>D3</n><m>M3:M4</m><s></s><cp>CP9:CP10</cp><bs>CP9</bs><pos>50:300</pos></d>\
<d><n>D4</n><m>M2</m><s>S4:S5</s><cp>CP7</cp><bs></bs><pos>250:450</pos></d>"

theRHW2PNet.controller._inter_domain="\
<l><n>R1</n><cp>CP5:CP8</cp><pos>400:150</pos></l>\
<l><n>R2</n><cp>CP9:CP6</cp><pos>50:150</pos></l>\
<l><n>R3</n><cp>CP10:CP7</cp><pos>120:400</pos></l>"
```

Figure 22– Configuration file related to the `Controller` module instance (excerpt)

The meaning of most of the tags used on the `_domain` string has been described in Chapter 5.

The meaning of the tags used in `_domain` parameter are the following: <*d*> and </*d*> specify a domain; the tags <*n*> and </*n*> enclose the name of the `Domain` module instance; <*m*> and </*m*> enclose the name of the masters belonging to the domain, which are separated by a colon; <*s*> and </*s*> tags are similar to the previous case but associated with slaves; <*cp*> and </*cp*> tags enclose the names of the `Connection_Point` module instances that are connected to the `Domain` module instance, the names must be separated by colon; <*bs*> and </*bs*> tags enclose the name of the `Connection_Point` module instance, which operates as a BS of a wireless domain. In the particular case of Figure 22, the second domain D2, is described by: ("*<d><n>D2</n><m>M1:M5</m><s>S1:S2:S3</s><cp>CP6:CP5</cp><bs></bs><pos>200:150</pos></d>*"), it is composed by two `Master` module instance (M1 and M5), and three `Slave` module instances (S1, S2 and S3) and this `Domain` module instance is connected to two `Connection_`

`Point` module instances (CP6 and CP5). The `Domain` module instance is depicted in the screen at position (200,150).

The parameter `_inter_domain` is a string that is similar to the `_domain` string. This string defines the repeater configuration, and the meaning of the tags is the following: <*r*> and </*r*> define a repeater; <*n*> and </*n*> enclose the name of the `ComFunc` module instance; between tags <*cp*> and </*cp*> and separated by a colon appear the names of the `Connection_Point` module instances; <*pos*> and </*pos*> are used to define the location of the repeater. The first repeater presented in Figure 6.4 (*"<r><n>R1</n> <cp>CP5:CP8</cp><pos>400:150 </pos></r>"*) is referred to as R1, it is composed by two `Connection_Point` module instances (CP5 and CP8) and is positioned at (400,150). Note that, this repeater interconnects domains D1 and D2.

The `Controller` module instance stores into internal variables, the structure of the network. By manipulating this information it changes the network configuration when wireless mobile stations (`Master` and `Slave` module instances) move between domains.

This information enables the `Controller` to set the LAS of all `Master` module instances as well as PS, NS and GAPL parameters. It also assigns the token frame to the master defined as DMM, thus, avoiding the need to perform the standard PROFIBUS network initialization procedure.

5.1.3. Domain

In spite of the OMNeT++ capacities, only one-to-one connections are supported. One-to-many and many-to-one connections can only be achieved using special purpose simple modules. Therefore, it was necessary to develop a simple module – the `Domain` module which is able to connect all stations in a domain and simulate a broadcast network. The connections are created and assigned dynamically enabling the support of mobility. In our model we assume that the propagation delay is ignorable. The transmission delay is simulated by the `Domain` module as a function of the `Baud_rate` parameter and the message length.

The parameters that are common to all modules connected to a domain are assigned to the `Domain` module and the `Controller` module instance performs the domain configuration and other module instance parameterization using this information.

Figure 23 presents the `Domain` simple module NED definition. The parameter `_medium` defines if the `Domain` module instance maps into wired (different to zero) or wireless (equal to zero) domain. Due to the use of different media in the network, the format of the wired and wireless frames is different. As an example, each DLL character can be coded using 8 or 11 bit, for wireless and wired frames, respectively. The wireless frames can also include additional preamble and header fields. The parameters `bitsPerChar`, `frameHeadLen` and `frameTailLen` are the number of bits per character, the number on the bits of the frame head and the number of the bit on the frame tail, respectively.

```
simple Domain
    parameters:
        Baud_rate:          numeric,
        bitsPerChar:        numeric,
        frameHeadLen:       numeric,
        frameTailLen:       numeric,
        G:                  numeric,
        HSA:                numeric,
        TTR:                numeric,
        TSL:                numeric,
        max_retry_limit:    numeric,
        _bem_type:          numeric,
        _bem_par1:          numeric,
        _bem_par2:          numeric,
        _bem_par3:          numeric,
        _bem_par4:          numeric,
        _beacon_len:        numeric,
        n_beacon:           numeric,
        beacon_gap:         numeric,
        _name:              string;
    gates:
        in:                 ctrl_gateIn;
        out:                ctrl_gateOut;
endsimple
```

Figure 23 – `Domain` module NED definition

The parameter G is the Gap Update factor. The HSA parameter defines the Highest Station Address in the domain. The TTR parameter is the Target Rotation Time (T_{TR}) of the token and TSL is the Slot Time (T_{SL}). The number of retries is defined by the max_retry_limit parameter. The parameters with _bem prefix are used to define the channel Bit Error Model (BEM) in use. In [12] we describe the BEM supported by these simulators.

Figure 24 presents part of the configuration file related to one instance of the Domain module. This Domain module instance is called D1 and maps a wireless domain operating at 2 MBits/s (Baud_rate parameter). Each frame has a head of 32 bits, no frame tail and each character is coded using 8 bits. The G is set to one, therefore the GAP Update mechanism is always active. The HSA can be set differently for each domain according to the highest address of the stations that can belong to its domain. The TTR and TSL parameters are set in bit times and represent the T_{TR} and T_{SL} PROFIBUS parameters, respectively.

Transmission errors are modelled using the Independent Channel Model [4] (_bem_type parameter equal to 1) with a bit error probability of 10^{-5} (0.00001) (_bem_par1 parameter).

During the mobility procedure a BS will transmit 14 Beacon frames with an interval of 25 µs between them and the length of each Beacon frame is equal to 10 bytes (Figure 6.6).

```
theRHW2PNet.domain[0]._name="D1"
theRHW2PNet.domain[0]._medium=0
theRHW2PNet.domain[0].Baud_rate=2000000
theRHW2PNet.domain[0].frameHeadLen=32
theRHW2PNet.domain[0].frameTailLen=0
theRHW2PNet.domain[0].bitsPerChar=8
theRHW2PNet.domain[0].G=1
theRHW2PNet.domain[0].HSA=5
theRHW2PNet.domain[0].TTR=300
theRHW2PNet.domain[0].TSL=115
theRHW2PNet.domain[0].max_retry_limit=1
theRHW2PNet.domain[0]._bem_type=1
theRHW2PNet.domain[0]._bem_par1=0.00001
theRHW2PNet.domain[0].beacon_len=10
theRHW2PNet.domain[0].n_beacon=14
theRHW2PNet.domain[0].beacon_gap=25us
```

Figure 24 – Configuration file related to Domain module instance (excerpt)

5.1.4. Master

A Master module is a compound module that maps a master station. It is composed by three modules: Master_PHY, Master_DLL and Msg_Stream. In each Master module instance there is one instance of Master_PHY and Master_DLL modules. The number of the Msg_Stream module instances can be from 1 up to 64. A Master module is connected to the Domain module through gates domain_gateIn and domain_gateOut. Master_PHY and Master_DLL model the PhL and DLL of the PROFIBUS protocol, respectively. The Msg_Stream module models the operation of the Application Layer (AL), therefore it can be configured to periodically request services from the DLL. These modules are hierarchically organized as illustrated in Figure 25.

As mentioned, compound modules are modules composed of one or more sub-modules. Any module type (simple or compound module) can be used as a sub-module. Further, sub-modules may use parameters of the compound module.

The compound module definition specifies how the gates of the compound module and its immediate sub-modules are connected. Connections that span multiple levels of the hierarchy are not allowed. This restriction enforces compound modules to be self-contained. These concepts are presented in the Master module NED definition depicted in Figure 26. In this definition some parameters are omitted since its definition is very long.

The address of the Master module instance is set using the TS parameter. The number of message streams is defined by the _num_streams parameter. This parameter is used to define the number of Msg_Stream module instances and also to specify the number of gates between the Master_DLL module and Msg_Stream module instances. The parameters with _pdf_tid1 prefix are related to T_{ID1} PROFIBUS DLL parameter.

Figure 27 shows part of the configuration file related to a `Master` module instance. The parameter `_pdf_tid1_type` is set to three meaning that the T_{ID1} duration evolves according to a Triangular PDF. A Triangular PDF requires three parameters. In this case, the value of T_{ID1} will be between 11 bit times (`_pdf_tid1_par1` parameter) and 100 bit times (`_pdf_tid1_par3` parameter) and the mode is 70 bit times (`_pdf_tid1_par2` parameter).

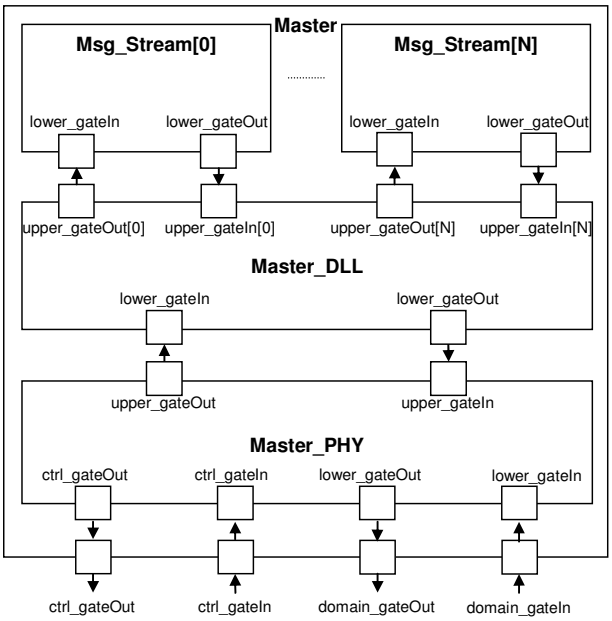

Figure 25 – OMNeT++ `Master` module composition

```
module Master
    parameters:
        TS:                 numeric,
        _num_streams:       numeric,
        _name:              string,
        _pdf_tid1_type:     numeric,
        _pdf_tid1_par1:     numeric,
        _pdf_tid1_par2:     numeric,
        _pdf_tid1_par3:     numeric,
        …
    gates:
        in:                 domain_gateIn,ctrl_gateIn,;
        out:                domain_gateOut,ctrl_gateOu;
    submodules:
        phy_layer:          Master_PHY;
        dll_layer:          Master_DLL;
                parameters:
                        TS=TS,
                        _num_streams=_num_streams,
                        _pdf_tid1_type=_pdf_tid1_type,
                        ..
                gatesize:
                        upper_gateOut[_num_streams],
                        upper_gateIn[_num_streams];

        stream:             Msg_Stream[_num_streams];
    connections nocheck:
            phy_layer.upper_gateOut --> dll_layer.lower_gateIn;
                phy_layer.upper_gateIn <-- dll_layer.lower_gateOut;
                phy_layer.lower_gateIn <-- domain_gateIn;
                phy_layer.lower_gateOut -->domain_gateOut;
                phy_layer.ctrl_gateIn <-- ctrl_gateIn ;
                phy_layer.ctrl_gateOut --> ctrl_gateOut;
            for i=0..._num_streams-1 do
                dll_layer.upper_gateOut[i] --> stream[i].lower_gateIn;
                dll_layer.upper_gateIn[i] <-- stream[i].lower_gateOut;
            endfor;

endmodule
```

Figure 26 – `Master` module NED definition

```
theRHW2PNet.master[2].TS=3
theRHW2PNet.master[2]._name="M3"
theRHW2PNet.master[2]._pdf_tid1_type=3
theRHW2PNet.master[2]._pdf_tid1_par1=11
theRHW2PNet.master[2]._pdf_tid1_par2=70
theRHW2PNet.master[2]._pdf_tid1_par3=100
```

Figure 27– Configuration file related to `Master` module instance (excerpt)

The following sections describe the implementation of each module that composes a `Master` module instance and their interactions.

Master_PHY

The `Master_PHY` module models the PhL of the PROFIBUS protocol. It represents the network interface of the `Master` module, it receives messages from a `Domain` or from a `Controller` module instance and passes the messages to the `Master_DLL` module and vice-versa. For that reason, this module is connected to the `Master` compound module through four gates (see Figure 5.10): `domain_gateIn`, `domain_gateOut`, `ctrl_gateIn` and `ctrl_gateOut`, the first two are related to `domain_con` connections and the last two are related `ctrl_con` connections. Figure 28shows the `Master_PHY` NED definition.

```
simple Master_PHY
    gates:
    in: lower_gateIn, ctrl_gateIn, upper_gateIn;
    out: lower_gateOut, ctrl_gateOut, upper_gateOut;
endsimple
```

Figure 28 – `Master_PHY` module NED definition

Master_DLL

The `Master_DLL` module is a simple module that it is directly connected to the `Master_PHY` and the `Msg_Stream` modules. It is connected to the `Master_PHY` module instance through `lower_gateIn` and `lower_ gateOut` gates and it is connected to *N* `Msg_Stream` module instances through 64 gates `upper_gateIn[x]` and `upper_gateOut[x]`, where x is a number between 1 and 64.

Figure 5.14 presents part of the `Master_DLL` NED definition. Its NED definition is very simple since most of its parameters are dynamically configured by the `Controller` module instance.

Msg_Stream

The `Msg_Stream` module models the typical behaviour of the AL. It can be configured to periodically request services from the `Master_DLL` module instance through the `lower_gateOut` gate. Each `Msg_Stream` module instance must be configured with the parameters necessary to build PROFIBUS messages. Figure 5.15 shows the `Msg_Stream` NED definition. The parameters `DA` and `SA` refer to `Destination Address` and `Source Address`, respectively. The local access address to the AL is defined in the `SAE – Source Address Extension –` and the remote access address to the AL is defined in the `DAE – Destination Address Extension`.

The parameter `DATA_UNIT` maps the content of a frame data field. For simplification reasons this parameter is a numeric data field. `Serv_class` parameter defines the priority (high or low) for the data transfer and the `service` parameter defines if a message maps into a `Send Data with No Acknowledge` (SDN) or a `Send and Request Data with Reply` (SRD) PROFIBUS service. If the `service` parameter value is set to 1, it means that it models a SDN service. In the case of being 3, it is a SRD service. The PROFIBUS SDA service can be modelled by using a SRD with a one byte response frame.

```
simple Master_DLL
    parameters:
        TS:                 numeric,
        _pdf_tid1_type:     numeric,
        _pdf_tid1_par1:     numeric,
        …
        _pdf_tid2_type:     numeric,
        _pdf_tid2_par1:     numeric,
        …
        _pdf_tsdr_type:     numeric,
        _pdf_tsdr_par1:     numeric,
        …
    gates:
        in:                 upper_gateIn[],lower_gateIn,;
        out:                upper_gateOut[],lower_gateOut,;
endsimple
```

Figure 29 – OMNeT++ `Master_DLL` module NED definition

The message generation can be active or inactive. If the value of the `_active` parameter is 0, then no messages are generated, otherwise messages are periodically generated.

```
simple Msg_Stream
    parameters:
        DA:                 numeric,
        DAE:                numeric,
        SA:                 numeric,
        SAE:                numeric,
        DATA_UNIT:          numeric,
        Serv_class:         numeric,
        service:            numeric,
        _active:            numeric,
        _deadline:          numeric,
        _output_color_red:  numeric,
        _output_color_green: numeric,
        _output_color_blue: numeric,
        _pdf_length_type:
        …
        _pdf_period_type:   numeric,
        …
        _pdf_offset_type:   numeric,
        …
    gates:
        in:                 lower_gateIn;
        out:                lower_gateOut;
endsimple
```

Figure 30 – `Msg_Stream` NED definition

Typically, a transaction (or message cycle) consists of the request or a send/request frame from the initiator and the associated response frame from the responder, especially for SRD. The response time of each transaction is computed from the time in which the request frame is queued on the DLL output message queue until it receives the response frame.

However, the response time of a transaction can be theoretically unlimited in error prone environments. In order to deal with transmission medium characteristics, real time systems must be provided with mechanisms to detect and handle these error situations [13]. In a communication system these mechanisms are implemented at all levels of the communication stack.

At the AL level the `Msg_Stream` module is provided with a parameter (the `_deadline` parameter) which is used to detect if a transaction is not concluded before the expiration of the deadline.

In our simulation model we consider that a transaction misses its deadline in two situations. First, when the response time of a transaction is higher than its deadline even when a valid response is obtained from the IDT BM_{ini}, this is illustrated on the left side of Figure 31. The second case is the most common case in which a deadline is considered missed when the response frame is not received within its deadline. This case is illustrated on the right side of Figure 31.

As mentioned, these simulators produce information enabling the display of the Gant diagrams concerning message transactions. To distinguish between the different message streams it is possible to assign a colour to each message stream using the parameters with `_output_color_red`, `_output_color_green` and `_output_color_blue`.

The length in bits, the period and offset of the first activation of the message streams can be assigned either using random or constant values.

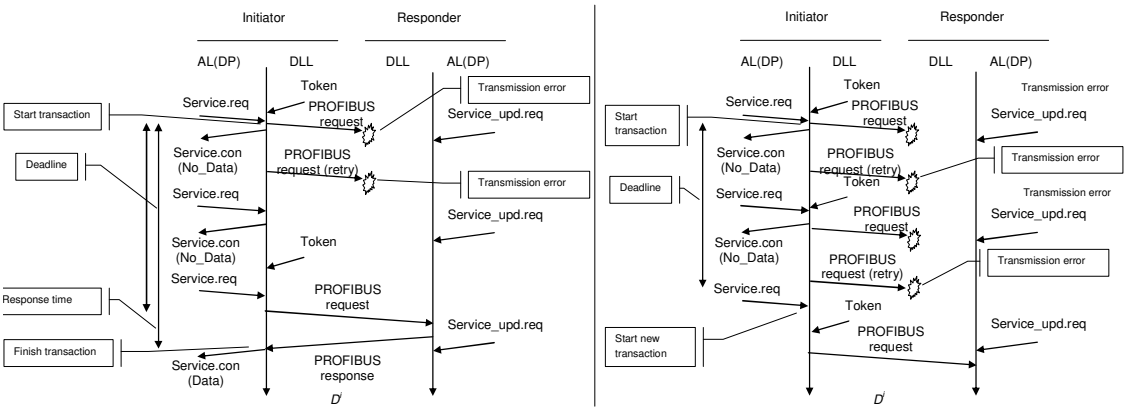

Figure 31– Deadline missing examples

Figure 32 depicts an example of an `Msg_Stream` module instance configuration. This message stream involves a `Master` module instance with address 3 (`SA`) and `Slave` module instance with address 46 (`DA`). The SAE and DAE have the same value, which is equal to 7.

```
theRHW2PNet.master[2].stream[1].DA=46
theRHW2PNet.master[2].stream[1].SA=3
theRHW2PNet.master[2].stream[1].DAE=7
theRHW2PNet.master[2].stream[1].SAE=7
theRHW2PNet.master[2].stream[1].DATA_UNIT=0
theRHW2PNet.master[2].stream[1].Serv_class=1
theRHW2PNet.master[2].stream[1].service=3
theRHW2PNet.master[2].stream[1]._active=1
theRHW2PNet.master[2].stream[1]._deadline=100ms
theRHW2PNet.master[2].stream[1]._pdf_length_type=0
theRHW2PNet.master[2].stream[1]._pdf_length_par1=15
theRHW2PNet.master[2].stream[1]._pdf_period_type=0
theRHW2PNet.master[2].stream[1]._pdf_period_par1=0.005
theRHW2PNet.master[2].stream[1]._pdf_offset_type=0
theRHW2PNet.master[2].stream[1]._pdf_offset_par1=0.0
theRHW2PNet.master[2].stream[1]._output_color_red=100
theRHW2PNet.master[2].stream[1]._output_color_green=255
theRHW2PNet.master[2].stream[1]._output_color_blue=0
```

Figure 32– `Msg_Stream` configuration parameters of a `Master`

As the `Serv_class` parameter is equal to 1 this is a high priority message stream using the SRD service (since the `service` parameter is equal to 3).

The period of this message stream is constant (since the `_pdf_period_type` parameter is equal to zero) at 0.005 s (`_pdf_period_par1` parameter). The length of the message is also constant (since the `_pdf_length_type` parameter is equal to zero) at 15 bytes (`_pdf_length_par1` parameter). The first message does not have any initial offset. The colour of this message stream on the Gant diagram is (100, 255, 0) in RGB notation.

5.1.5. Slave

A `Slave` is a compound module which maps into a standard PROFIBUS slave station. It is structured similarly to the `Master` module. The `Slave_PHY` module is equal to the `Master_PHY` module. The `Msg_Stream` module is the same module used by the `Master` module and is used for the same purpose, although in the case of a `Slave` module it generates periodical response messages. The `Slave_DLL` module is a simple module, which maps the PROFIBUS DLL of a slave.

In each `Slave` module instance there is one instance of `Slave_PHY` and `Slave_DLL` modules. The number of the `Msg_Stream` module instances can be from 1 up to 64. As shown in Figure 33the `Slave` module structure is very similar to the `Master` module structure presented in Figure 25.

Since the `Master_PHY` and `Msg_Stream` modules were already described in Section 5.1.4, in this sub-section only the `Slave_DLL` module will be described.

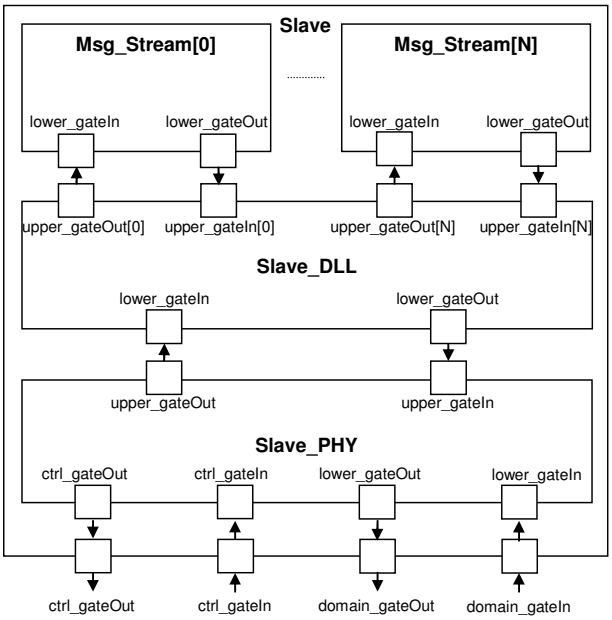

Figure 33 – OMNeT++ `Slave` module

Figure 34 presents the `Slave_DLL` NED definition. The address of `This Station` is set on its `TS` parameter. The T_{SDR} parameter is assigned according to the mode, defined in Section 5.2 for these types of parameters (which can receive values from PDF functions).

```
simple Slave_DLL
    parameters:
        TS:                 numeric,
        _pdf_tsdr_type:     numeric,
        _pdf_tsdr_par1:     numeric,
        …
    gates:
        in:                 upper_gateIn[],lower_gateIn;
        out:                upper_gateOut[],lower_gateOut;
endsimple
```

Figure 34– `Slave_DLL` module NED definition

5.1.6. Parameters `_location_vector` and `_is_mobile_station`

Besides the parameters referred in previously, `Master` and `Slave` modules have two more parameters. One called `_location_vector` and the other called `_is_mobile_station`, these parameters are highlighted in Figure 35. The `_location_vector` is a string which defines the location of each `Master` and `Slave` module instance during time. In order to limit the size of the configuration files used, the `_location_vector` parameter is written in a compact format. Each location is represented by a tuple (*n_mob*,*Dx*), where *n_mob* represents the number of mobility procedures during which the `Master` or `Slave` module instance will stay on domain *Dx*.

The `_is_mobile_station` parameter is used to define if a `Master` or a `Slave` module instance models a mobile station (assigned with one) or not (assigned with zero).

Figure 35 depicts part of the configuration file related to a `Master` module instance, which models a wireless mobile station called M3. This station stays in domain D1 for five mobility procedures, and then it changes to domain D3 where it will stay for another 10 mobility procedures. This sequence of events repeats itself until the end of the simulation.

```
theRHW2PNet.master[2]._name="M3"
theRHW2PNet.master[2]._is_mobile_station=1
theRHW2PNet.master[2]._vector_location="5,D1:10,D3:"
```

Figure 35– Configuration file related to the `Master` module instance of the RHW2PNetSim (excerpt)

In order to set the `_location_vector` parameter according to the radio channel quality and the mobility of wireless mobile station the Mobility Simulator (MSim) has been developed. This simulator models the radio wave propagation according to the Log-normal Shadowing model [14] and the mobility of wireless mobile stations. A detailed description of this simulator is found in [15].

5.1.7. Repeater Architecture

There is no module called repeater, the repeater is in fact an abstraction, since its operation is supported by three module instances, one instance of the `ComFunc` module and two of `Connection_Point` module (Figure 36)**.**

The `ComFunc` module instance establishes connections between the `Connection_Point` module instances that belong to the repeater through the `com_func_con` connections. The `Connection_Point` module instances establish the connections to the `Domain` module instances by the `repeater_con` connections.

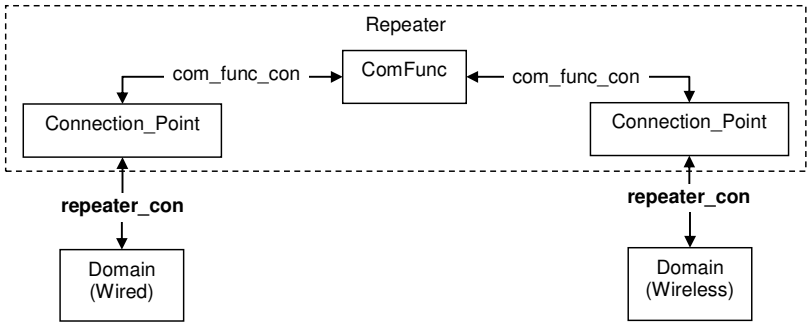

Figure 36– Repeater's module instances and their connections

ComFunc

`ComFunc` is a simple OMNeT++ module. The NED definition of the `ComFunc` module is given in Figure 37. The main function of this module is to model the internal `relaying delay` – t_{rd} .

Frames relayed by the repeater are delayed by this module. The delay value is assigned to the parameters with the prefix `_pdf_delay`.

```
simple ComFunc
    parameters:
        _pdf_delay_type:    numeric,
        _pdf_delay_par1:    numeric,
        _pdf_delay_par2:    numeric,
        _pdf_delay_par3:    numeric,
        _name:              string;
    gates:
        in:                 ctrl_gateIn;
        out:                ctrl_gateOut;
endsimple
```

Figure 37– `ComFunc` module NED definition

Figure 38 presents part of a configuration file related to a `ComFunc` module instance. This instance is called R1 and the delay introduced is equal to 30 µs. The parameter `_pdf_delay_type` defines if the delay is constant (0) or random according to a PDF, see [™®] for details.

```
theRHW2PNet.repeater[0]._name="R1"
theRHW2PNet.repeater[0]._pdf_delay_type=0
theRHW2PNet.repeater [0]._pdf_delay_par1=0.00003
```

Figure 38– Configuration file related to one `ComFunc` module instance (excerpt)

Connection_Point

`Connection_Point` is a simple OMNeT++ module. This module establishes the connections between the `Domain` module instances and assures that a frame is transmitted without time gaps. Additionally, the BS functions are also modelled in this module like for instance, sending `Beacon` frames during the mobility procedure. Figure 39 shows the `Connection_Point` connections**.**

Figure 40 presents the NED configuration of the `Connection_Point` module. The inactivity time between two consecutive frames (T_{IDm} described in Section 2.3.4) can be assigned in a stochastic way. For this reason, the timing delay can be assigned by four parameters. The parameters related to the T_{IDm} have the `_pdf_tidm` prefix.

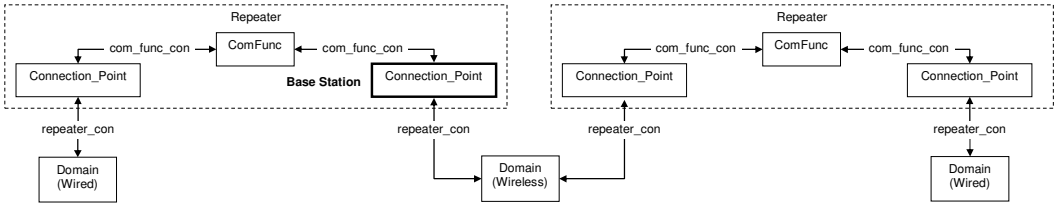

Figure 39– Connection_Point module connections

```
simple Connection_Point
    parameters:
        _pdf_tidm_type:     numeric,
        _pdf_tidm_par1:     numeric,
        _pdf_tidm_par2:     numeric,
        _pdf_tidm_par3:     numeric,
        _name:              string;
    gates:
        in:                 com_func_gateIn,domain_gateIn;
        out:                com_func_gateOut,domain_gateOut;
endsimple
```

Figure 40– `Connection_Point` module NED definition

Figure 41 presents part of a configuration file related to a `Connection_Point` module instance. This instance is called CP8. Since the parameter `_pdf_tidm_type` is set to zero, then the inactivity period between the two consecutive frames is constant and equal to 100 bit times.

```
theRHW2PNet.cp[3]._name="CP8"
theRHW2PNet.cp[3]._pdf_tidm_type=0
theRHW2PNet.cp[3]._pdf_tidm_par1=100
```

Figure 41 – Configuration file related to the `Connection_Point` module instance (excerpt)

In order to set the `_location_vector` parameter according to the radio channel quality and the mobility of wireless mobile station the Mobility Simulator (MSim)[15] has been developed. This simulator models the radio wave propagation according to the Log-normal Shadowing model [14] and the mobility of wireless mobile stations.

5.2. PROFIBUS DLL Basic Implementation

In this Section an overview of the PROFIBUS DLL basic implementation. In a standard PROFIBUS a slave state machine is composed by two states: OFFLINE and PASSIVE_IDLE. In our implementation of both simulators, the slave state machine only uses one state, the PASSIVE_IDLE state.

A slave does not have initiative, it only responds to requests addressed to it. The Slave_DLL behaviour depends on the kind of the received frame. In the implementation of both simulators, a slave is only able to receive SDN, SRD and FDL_Request_Status request frames.

The operation mode of a standard master PROFIBUS DLL is supported by a state machine composed by 10 states [5]. For simplification reasons, in both implementations only 8 states are considered.

According to the PROFIBUS protocol, after turning on the power of a master station it will go into the LISTEN_TOKEN state in order to generate the List of Active Stations (LAS) and GAP List (GAPL). However, all Master module instances start in the ACTIVE_IDLE state with all configurations and parameterization performed by the Controller module instance, at the simulation setup. In order to start the network simulation operation, the Controller module instance sends a token frame to the Domain Mobility Manager (DMM) of each domain. Then the state machine of the Master module instances evolves to the USE_TOKEN state.

Figure 42 presents the Master state machine diagram related to the implementation of the PROFIBUS DLL in both simulators.

In this state machine diagram an oval shape represents a state and an arrow a transition. For better identification, within of the oval shape the state identification is written and each transition is identified by a number.

In the description of the Master state machine diagram it is assumed that the Time-Out Time (T_{TO}) and the Slot Time (T_{SL}) related timers are automatically started and stopped in the following situations. The PROFIBUS protocol, defines that when a master frame's last bit is transmitted or when a master frame's last bit is received a timer is loaded with the T_{TO} value and is started (hereafter referred to as T_{TO} timer). The T_{TO} timer is stopped after receiving the first bit of the following frame. The value of this timer is set according to the Eq. 2. Whenever a request frame's last bit is transmitted by an initiator that requires either a response or an acknowledgement a timer is loaded with T_{SL} parameter value and is started (hereafter referred to as T_{SL} timer).

When a Master is in the ACTIVE_IDLE state, 3 transitions (1, 19 and 13) are possible. Transition 1 is triggered either by the reception of a valid token frame from its Previous Station (PS) (see Section 5.2.2 for more details), or when T_{TO} expires.

Transition 19 occurs when a Master receives either a valid frame (without bit errors) not addressed to it or if the frame is valid and addressed to it but is not a token frame, for example an FDL_Request_Status frame (more details are given in Section 5.2.5). If the received frame is an invalid frame (containing bit errors) the Master continues in the same state.

A Master in the ACTIVE_IDLE state continually analyses all received frames. If a token frame is received when This Station (TS) is "skipped" (i.e., the address of TS lies within the address range spanned from the sender address to the receiver address in the token frame), then it removes itself from the logical ring and evolves to the LISTEN_TOKEN state (transition 13).

When the Master is in the USE_TOKEN state, i.e., when it holds the token frame, it behaves according to the message dispatching procedure (a detailed description of this procedure is presented in Section 5.2.3). A Master in the USE_TOKEN state can perform one of 4 transitions (2, 3, 5 and 7). The Master stays on the same state (transition 2) if it transmits a frame that does not need to receive a response frame (e.g., when using the SDN service, see Section 5.2.6 for more details). It changes to the AWAIT_DATA_RESPONSE state (transition 3) if it sends a message that requires a response frame (e.g., when using the SRD service, see Section 5.2.6) and returns to the USE_TOKEN state when it receives a response frame or the Slot Time (T_{SL}) expires (transition 4).

Transition 5 occurs, when the Master transmits an FDL_Request_Status frame and evolves to the AWAIT_STATUS_RESPONSE state. The Master returns to the USE_TOKEN state (transition 6) when it receives either a response frame or the T_{SL} expires (in Section 5.2.5the Gap update procedure is described in detail).

From the USE_TOKEN state it changes to the PASS_TOKEN state (transition 7) when its Token Holding Time (T_{HT}) expires. The T_{HT} is computed at token frame reception according to Eq. 1.

The transitions 8, 9, 10 and 11 are handled by the pass token procedure (a detailed description of this procedure is presented in Section 5.2.4).

From the PASS_TOKEN state it evolves to the CHECK_TOKEN_PASS state (transition 8) after transmitting a token frame to its `Next Station` (NS). In the CHECK_TOKEN_PASS state 2 transitions (9 and 12) can occur. If it detects a valid frame in its domain then it changes to the ACTIVE_IDLE state (transition 12). Otherwise, if after expiring T_{SL} no activity is detected it returns to the PASS_TOKEN state (transition 9) and stays into these two states (PASS_TOKEN and CHECK_TOKEN_PASS) until passing the token to another station in its LAS or to itself (transition 11).

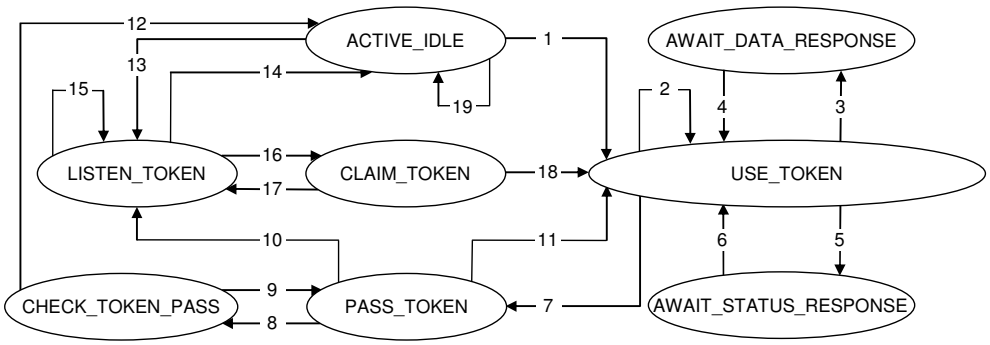

Figure 42– `Master` state machine diagram

In order to detect a defective transceiver when a `Master` is transmitting a token frame, in the TOKEN_PASS state, it also receives the token frame. If it detects a difference between the transmitted and received frame it waits, in the CHECK_TOKEN_PASS state, T_{SL} for activity from its NS. If no activity is detected after expiring T_{SL}, it again transmits the token frame, if it again detects a difference between the transmitted and received frames its state machine evolves to the LISTEN_TOKEN state (transition 10). This process is designated as "heardback removal" in [4].

Whenever a `Master` evolves to the LISTEN_TOKEN state the LAS is cleared and it starts listening on the medium for at least two successive identical token cycles. During this time it is not allowed to send or respond to data frames or to accept the token, but it builds the LAS. After that, its state machine evolves to the ACTIVE_IDLE state (transition 14).

The `Master` state machine evolves to the CLAIM_TOKEN state from the LISTEN_TOKEN state when the T_{TO} timer expires (transition 16). In this case there is the need to recover the token (Section 5.2.1 presents more details about this procedure). In this procedure it transmits two token frames addressed to itself and if no difference between transmitted and received frames occur its state machine evolves to the USE_TOKEN state (transition 18). Otherwise, the state machine evolves to the LISTEN_TOKEN state (transition 17) as a consequence of "heardback removal".

5.2.1. Token Recovery Procedure

In the PROFIBUS protocol, a token lost is detected when a master does not detect any network activity for a time defined by its `Time-Out Time` (T_{TO}) parameter (which is set by Eq. 2).

A timer is loaded with T_{TO} parameter value and is started in two situations. First, when the frame transmitter transmits the frame's last bit. Second, when a master receives frame's last bit. The timer is stopped when the first bit of the following frame is received.

When T_{TO} timer expires and the `Master` module instance is in the ACTIVE_IDLE state it starts performing message cycles according to the message dispatching procedure (described in Section 5.2.3). But if it is in the LISTEN_TOKEN state, then it evolves to the CLAIM_TOKEN state and the token recovery procedure starts. This procedure has two objectives. First, recovering the token frame and, second to reinitialize the logical ring.

Note that, when a `Master` evolves to LISTEN_TOKEN state all `List of Active Stations` (LAS) entries are deleted (Figure 43).

```
1.  handleSelfMessage(msg)
2.  {
3.  switch (getAction(msg)) {
4.        case TTO_TIMEOUT:
5.                  switch (state) {
6.                            case ACTIVE_IDLE:
7.                                    messageDispacthing();
8.                            end;
9.                            case LISTEN_TOKEN:
10.                                   state=CLAIM_TOKEN;
11.                                   tokenRecovery();
12.
13.                           end;
14.                 }
15.       end;
16.       ...
17.   }
18. }
```

Figure 43– `handleSelfMessage(msg)` function, pseudo-code algorithm

In order to recover the token and reinitialize the logical ring the `Master`, which is in the CLAIM_TOKEN state, transmits two token frames addressed to itself (Figure 44). The pass token procedure will be described in Section 5.2.4. In this way the token frame is recovered. After that, every `Master` will be joining to the logical ring using the GAP update procedure (described in Section 5.2.3).

Note that, when a `Master` transmits a token addressed to itself, all `Masters` that are not in the LISTEN_TOKEN state evolves to that state, since they are "skipped" of the logical ring.

```
1.  tokenRecovery()
2.  {
3.  passToken(TS);
4.  passToken(TS);
5.  state=USE_TOKEN;
6.  messageDispacthing();
7.  }
```

Figure 44– `tokenRecovery()` function, pseudo-code algorithm

5.2.2. Token Reception Procedure

The token frame is passed between masters in ascending Medium Access Control (MAC) address order. The only exception is that to close the logical ring the master with the `Highest Station Address` (HSA) must pass the token frame to the master with the lowest one. Each master knows the address of its `Previous Station` (PS), the address of the `Next Station` (NS) and its own address (`This Station` (TS)).

When a master receives a token frame addressed to itself from a master registered in the LAS as its PS then this master is said to be the token owner. On the other hand, if a master receives a token frame from a master, which is not its PS, it shall assume an error and will not accept the token frame. However, if it receives a second subsequent token frame from the same master, it shall accept the token frame and assumes that the logical ring has changed. In this case, it updates the original PS value by the new one and updates the LAS and the `Live List` (LL).

Figure 45 illustrates the token reception procedure. A token frame is discarded when it is an erroneous frame or if it is not addressed to the `Master`. If the token frame is addressed to the `Master` and it does not contain bit errors, then the `Master` behaviour depends on the token frame transmitter (i.e., if the token frame `Source Address` (SA) is registered as its PS).

If the token frame transmitter is registered as its PS, it evolves to the USE_TOKEN state, calculates the `Token Holding Time` (T_{HT}) according to the Eq. 1, sets to false `GAP_Turn` variable and then the token reception procedure ends. Otherwise, the received token frame is discarded when a `Master` token frame sender is not its PS. However, if it receives a second token frame from the same `Master`, then it updates the original PS value with the new one and updates its LAS.

In the same way it evolves to the USE_TOKEN state, calculates the new T_{TH} sets to false the `GAP_Turn` variable and then the token reception procedure ends. According to the PROFIBUS DLL only one GAP Update procedure can be performed per token visit, the `GAP_Turn` variable is used to

avoid that more than one GAP Update procedure is performed when a `Master` is holding the token frame.

The execution of the token reception procedure forces the execution of the message dispatching procedure described in the Section 5.2.3.

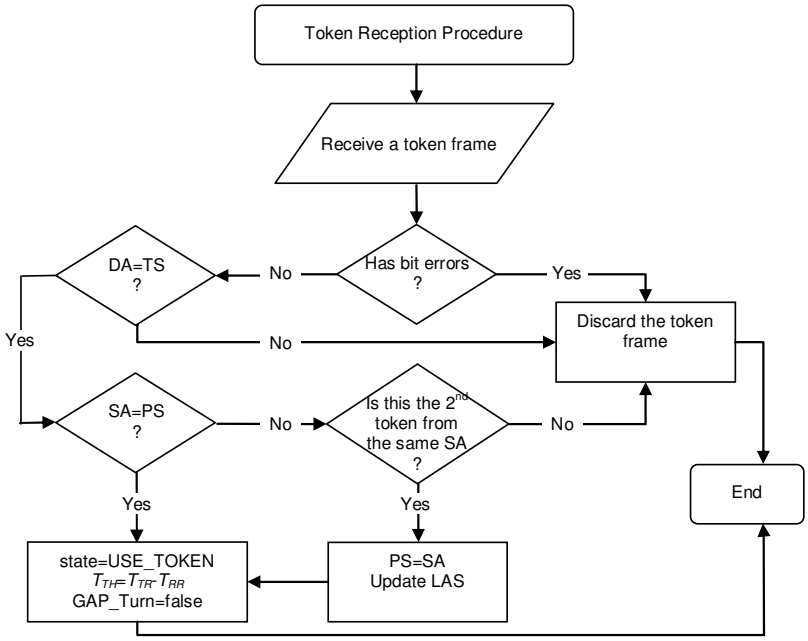

Figure 45 – Token Reception procedure

5.2.3. Message Dispatching Procedure

Figure 46 presents the message dispatching procedure when a `Master` holds the token frame. This procedure is repeatedly performed until the `Master` expires T_{TH} and the pass token procedure is executed.

At token frame reception, the period during which the `Master` is allowed to perform messages cycles (T_{TH}) is computed according to the Eq. 1.

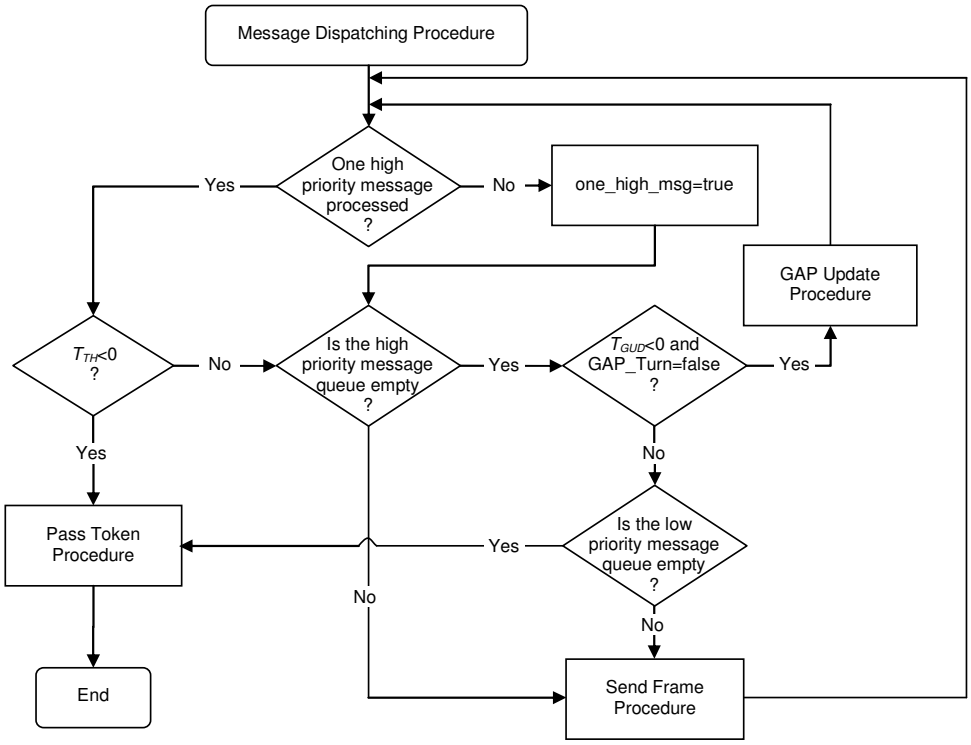

Figure 46 – Message dispatching procedure

Independently of the T_{TH} value a `Master` is allowed to transmit at least one high priority message. After, high priority message will be processed while $T_{TH} > 0$. When there are no more high priority messages to dispatch then low priority messages and GAP update related messages can be transmitted.

The GAP update procedure is triggered when the `Gap Update Timer` (T_{GUD}) expires and the $T_{TH} > 0$. If $T_{TH} < 0$, the GAP update procedure is postponed for the next token holding period. However, only one GAP update procedure is performed per token visit.

It should be pointed out that once a high or low priority message cycle or GAP update procedure is started, it is always completed (it is not pre-empted), including any retry (or retries), even if $T_{TH} < 0$. When $T_{TH} < 0$ or when the output message queues are empties, the `Master` passes the token frame to another station.

In this section, a high level message dispatching mechanism was described. The pass token, send frame and GAP Update procedures are detailed in the following sections.

5.2.4. Pass Token Procedure

The PROFIBUS protocol defines that token frames are passed between masters in ascending MAC address order. The only exception is that to close the logical ring the master with the HSA must pass the token to the master with the lowest one. Each master knows the address of the PS, the address of the NS and its address (TS) as well.

When T_{TH} expires or when no more messages are available on the queues (low and high priority), a master passes the token frame. If, after transmitting the token frame and after the expiration of `Slot Time` (T_{SL}) timer the token transmitter detects bus activity, it assumes that its NS owns the token and is performing message cycles. Otherwise, if the token transmitter does not recognize any bus activity within the T_{SL}, it re-sends the token frame and waits another T_{SL}. It assumes that its NS owns the token frame thereafter, if it recognizes bus activity within the second T_{SL}.

If, after the second retry, there is no bus activity, the token transmitter tries to pass the token to the next master on its LAS. It continues repeating this procedure until it has found a successor from its

LAS. If it does not succeed, the token transmitter assumes that it is the only one left in the logical ring and transmits the token frame to itself.

In order to detect a defective transceiver when a master is transmitting a token frame it reads back from medium all transmitted bits. If it detects a difference between the transmitted and received bits it waits T_{SL} for any activity from NS. If no activity is detected after expiring T_{SL}, it transmits again the token frame, if an error occurs, then it removes itself from the logical ring.

Figure 47 depicts the token pass procedure. The first step is to evolve the `Master` from the USE_TOKEN state to the PASS_TOKEN state and sets the `retry_counter` variable to zero. After that, it builds the token frame addressed to its successor (NS) and transmits the token frame. If the token frame received is equal to the token frame transmitted then it evolves to the CHECK_TOKEN_PASS state. If after transmitting the token frame and before the expiration of the T_{SL}, the `Master` receives a frame, it assumes that its NS owns the token and that it is executing message cycles, and evolves to the ACTIVE_IDLE state. If the `Master` does not receive a frame within the T_{SL}, it returns to the PASS_TOKEN state and it repeats the transmission of the token frame (its state machine evolves again to the CHECK_TOKEN_PASS state) for the last time (`retry_counter` = 2) and waits another T_{SL}. If it receives a frame within the second T_{SL}, it assumes a correct token frame transmission. Otherwise, it continues repeating this procedure until it has found a successor from its LAS (`getNS()`). If it does not succeed, it transmits the token frame to itself.

At the first time that the received token frame is different from the transmitted frame it transmits again the token frame. At the second time it evolves to the LISTEN_TOKEN state.

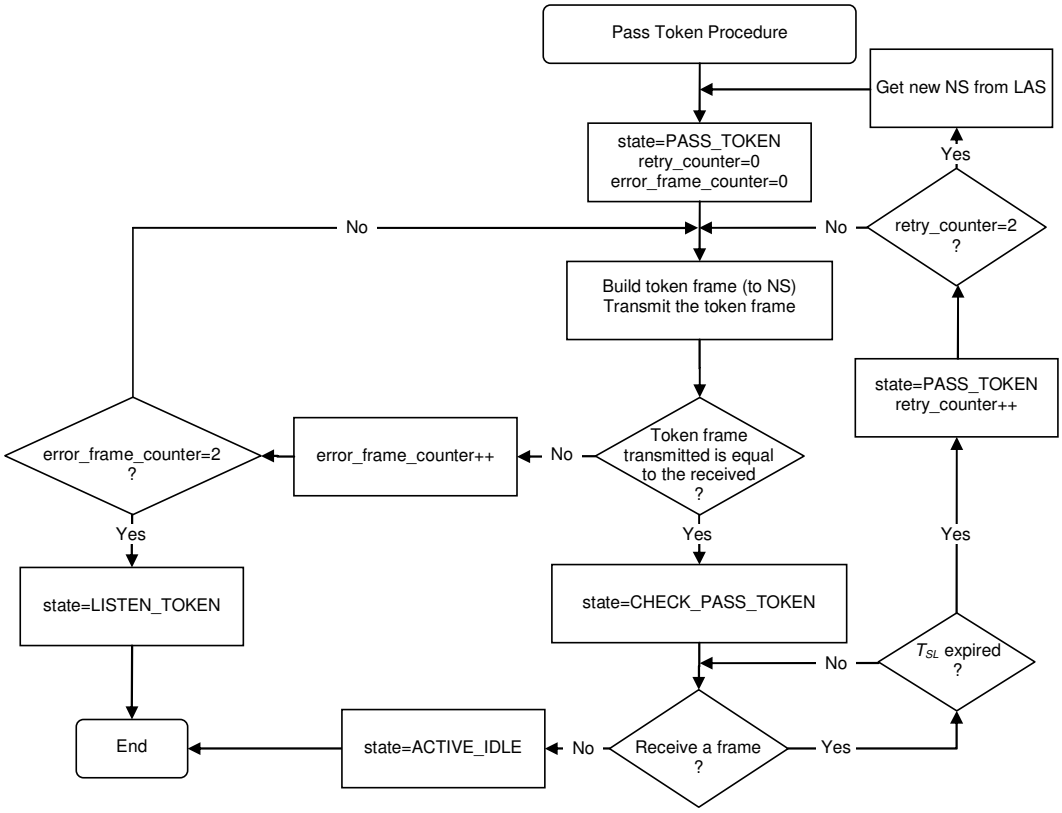

Figure 47 – Pass Token Procedure

5.2.5. GAP Update Procedure

Each master in the logical ring is responsible for the addition and removal of masters that have addresses between TS and NS. This range of addresses in the logical ring is referred as GAP, whereas the list containing the status of all stations in the GAP is called `GAP List` (GAPL).

Each master in the logical ring examines its GAP periodically in the interval given by the T_{GUD} timer. Its expiration indicates the moment for GAP maintenance. GAP addresses are examined in ascending order, except the GAP addresses which surpasses the HSA, i.e., the HSA and address 0 are not used by a master station. In this case the procedure is continued at address 1 after checking the HSA. If a station acknowledges positively with the state `Not_Ready_to_Enter_Logical_Ring` or `Slave_Station`, it is accordingly marked in the GAPL and the next address is checked. If a station answers with the state `Ready_to_Enter_Logical_Ring`, the token frame holder changes its GAPL and LAS accordingly, as well as its NS and passes the token frame to the new NS.

This is accomplished by examining at most one address per token cycle by means of sending an `FDL_Request_Status` once T_{GUD} has expired. After a complete GAP check, which may last several token rotations, the timer T_{GUD} is loaded with the value resulting from the multiplication of the T_{TR} by the `GAP Update Factor` (`G`). `G` represents the number of tokens rounds between GAP maintenance. Upon receiving the token, a GAP maintenance cycle starts immediately after all queued high priority message cycles have been processed and if there is still T_{TH} available. Otherwise, the GAP maintenance is postponed to the next token reception. Note that `FDL_Request_Status` messages have higher priority than low-priority messages used in PROFIBUS, but lower than high priority messages.

Send FDL_Request_Status Procedure

Figure 48 presents the send FDL_Request_Status procedure. The first step is to evolve the `Master` state machine from the USE_TOKEN to the AWAIT_STATUS_RESPONSE state. After that, it builds the `FDL_Request_Status` frame addressed to the selected station and transmits it. Thereafter, it waits for a valid (without bit errors) response frame from the addressed station within the T_{SL}. If either T_{SL} expires or the received response frame is an invalid, the `Master` evolves to the USE_TOKEN state and then the procedure ends.

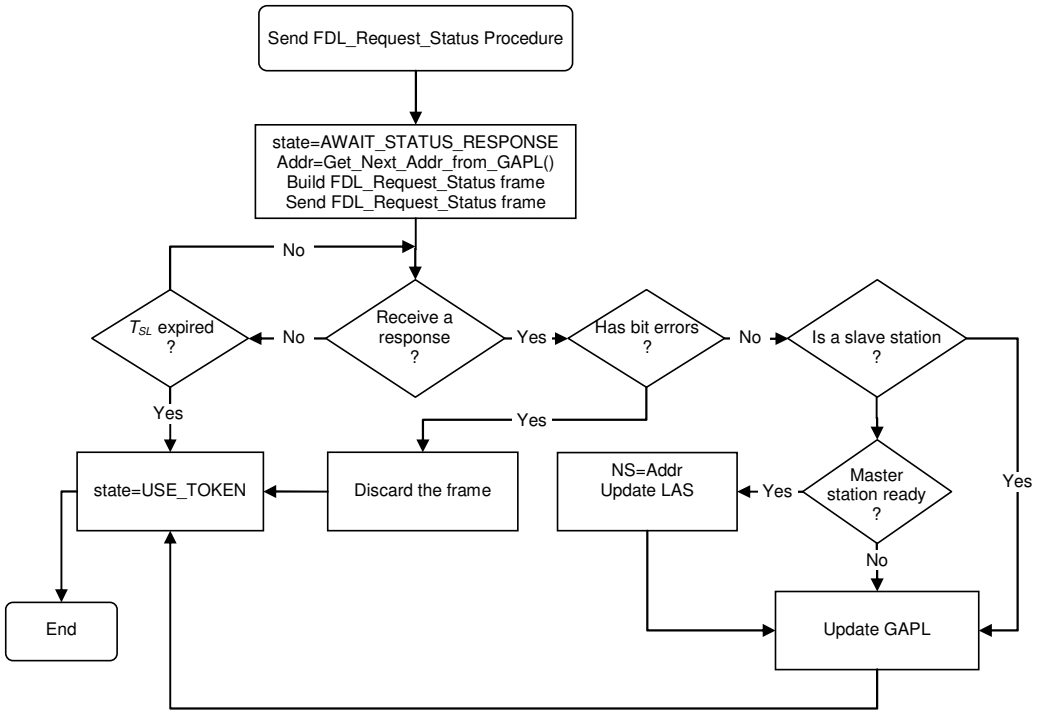

Figure 48 – Send FDL_Request_Status procedure

If a valid response frame was received from the addressed station the message is parsed. If the responder is a master and its state is `Ready_to_Enter_Logical_Ring` the `Master` changes the NS and updates the LAS and GAPL. Otherwise, it updates only the GAPL. In any cases, the `Master` evolves to the USE_TOKEN state.

Receive FDL_Request_Status Procedure

Figure 49 presents the procedure when a station receives an `FDL_Request_Status` frame. The frame is discarded if it contains bit errors or if is not addressed to it. Otherwise, a response frame containing the state of the responder is transmitted. In spite of, PROFIBUS DLL defines three states in our simulator only two were implemented: `Slave_Station`, if it is a slave station, and `Ready_to_Enter_Logical_Ring`, for the master stations.

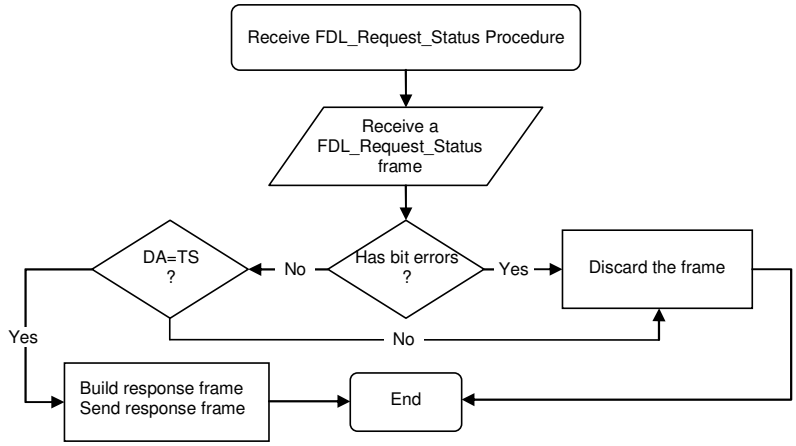

Figure 49 – Receive FDL_Request_Status procedure

5.2.6. Send Frame Procedure

The PROFIBUS DLL protocol defines four data transfer services. The `Send Data with Acknowledge` (SDA) service, which allows an initiator to send a message and immediately receive the confirmation. The `Send Data with No Acknowledge` (SDN) is an unacknowledged service. The `Send and Request Data with Reply` (SRD) is based on a reciprocal connection between an initiator and a responder and requires either an acknowledgement or a response. The `Send and Request Data with Reply` (CSRD) is a cyclic service (based on the acyclic SRD). In this simulator only the SDN and SRD services were implemented. The SDA service was implemented based on SDR service, just by adequate the setting of the request and response frame sizes.

The `Msg_Stream` module emulates the behaviour of an AL protocol. Periodically it produces messages which are passed to the `Master_DLL` module and forwarded to `DLL` module, which stores the message in its output message queues as a function of the message priority.

When a `Master` has the right to access the medium, i.e., when it holds the token frame, it pops messages from its output message queues and it checks which service will be used. If it is a SDN, then after transmitting the message, it can schedule a new action according to the message dispatching procedure. Otherwise, it has to wait for the response or the T_{SL} expiration (Figure 50).

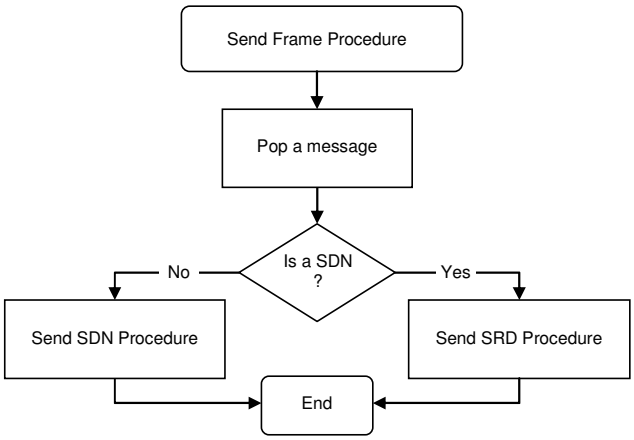

Figure 50 – Send Frame Procedure

Send SDN Procedure

A SDN is an unacknowledged service, therefore when an initiator sends a SDN frame, it will not receive a response or acknowledge from the responder.

Figure 51 illustrates a SDN transaction between a `Master` and a `Slave`. In a `Master`, the `Msg_Stream` emulates the behaviour of an AL protocol, periodically producing messages which are passed to the `Master_DLL`. The `Master_DLL` stores the message in its output message queues (high or low) according to the message priority. In a `Slave`, a `Msg_Stream` has a similar behaviour, but in this case, the `Slave_DLL` only refreshes the content of the variables modelled by the `Msg_Stream` module.

When a `Master` holds the token frame it processes messages cycles according to message dispatching procedure described in Section 5.2.3. If a `Master` has messages in its output queues it pops a message, builds a frame, passes to the `Master_PHY` and then sends to the `Domain` module to which it is connected. The `Domain` broadcasts the frame to every `Master` and `Slave` connected to it.

The `Slave_PHY` receives the frame from `Domain` and passes to the `Slave_DLL`. The `Slave_DLL` matches the frame information (`Destination Address` (DA), SA `Destination Address Extension` (DAE) and `Source Address Extension` (SAE)) with its configured message streams. If it succeeds then it registers the information about this transaction for output analyses and discards the frame. Otherwise, the `Slave_DLL` discards the frame.

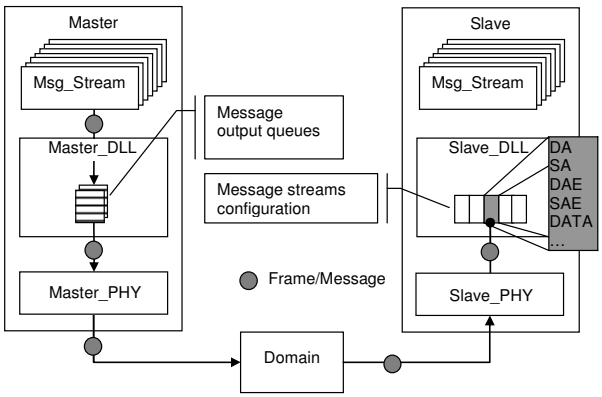

Figure 51 – SDN transaction schema between master and slave

Figure 52 presents the send SDN procedure. The `Master` pops a message from message output queues, builds a frame and then transmits it. Thereafter, `Master` waits T_{ID2} before performing another

action according to the message dispatching procedure. Note that, it stays on the same state (USE_TOKEN).

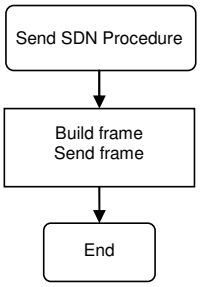

Figure 52 – Send SDN Procedure

Receive SDN Procedure

Figure 53 presents the receive SDN procedure. At frame reception the `Slave_DLL` starts by checking if it is a valid frame or not. The frame is discarded if it is an invalid frame. Otherwise, it checks if it is addressed to it or not (DA=TS). If not, the frame is discarded. If it is, the information about this transaction is stored for output analysis and the frame is discarded.

Send SRD Procedure

The SRD is based on a reciprocal connection between an initiator and a responder and requires either an acknowledgement or a response from the responder. Using this service, the initiator is able to send data in the request frame and receive data, from the addressed station, in the response frame.

Figure 54 illustrates the transaction schema between a `Master` and a `Slave`. When a `Master` holds the token frame its `Master_DLL` pops a message from one of its message output queues, builds a frame and passes it to the `Master_PHY` which sends to the `Domain` to which it is connected. The `Domain` broadcasts the frame to every `Master` and `Slave` connected to it.

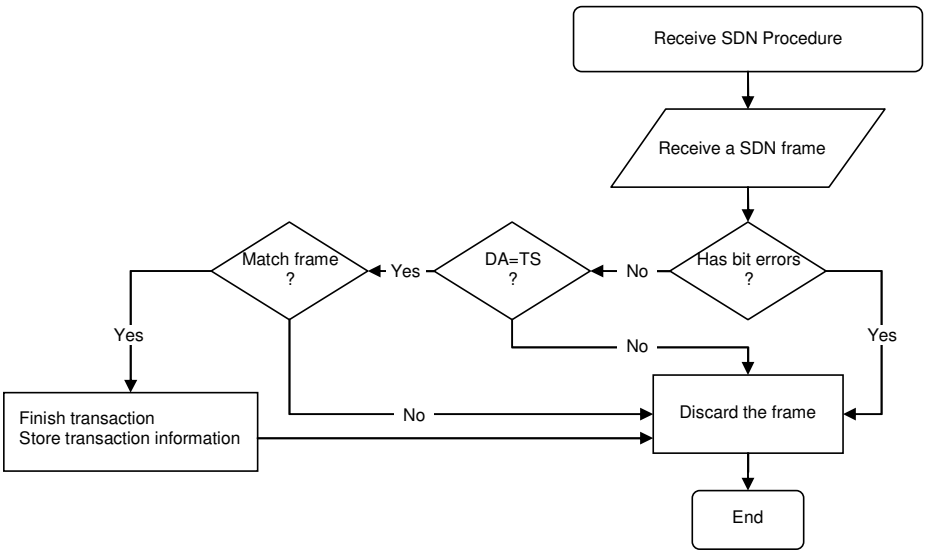

Figure 53 – Receive SDN Procedure

The `Slave_DLL` receives the frame from `Slave_PHY` and tries to determine if there is a match between the frame and a message stream configuration. If it finds a match, then it builds a response

frame and passes it to the `Slave_PHY` that sends it to the initiator through the `Domain`, otherwise the frame is discarded. The `Domain` broadcasts the frame to all `Master` and `Slave` module instances connected to it. The `Master_PHY` passes the response frame to the `Master_DLL`. If it is the addressed to the `Master`, it stores the information about this transaction and then discards the frame. If is not addressed to it, then the frame is discarded.

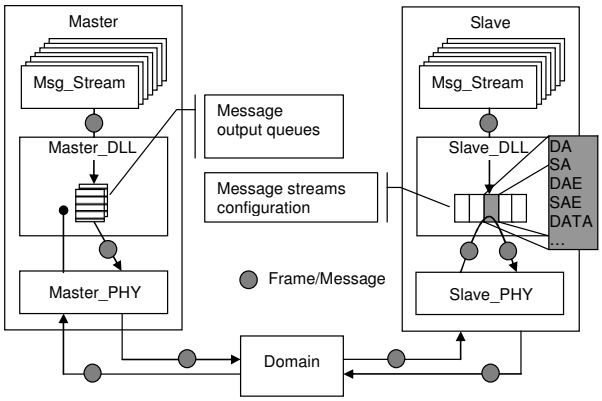

Figure 54 – SRD transaction schema between Master and Slave

Figure 55 presents the send SDR procedure. The first step is to evolve the `Master` state machine from the USE_TOKEN to the AWAIT_DATA_RESPONSE state and set the `retry_ counter` variable to zero.

After that, the `Master` pops a message from its message output queue, builds a frame and transmits it. After, it waits for the reception of a response frame. If an invalid response frame is received (with bit errors) then it is discarded and the `retry_counter` variable is increased. The `retry_counter` variable is also increased if no frame is received within the T_{SL}. When the `retry_counter` variable reaches the `max_retry_limit` (a `Master` parameter) limit then the `Master` state machine returns to the USE_TOKEN state.

Receive SRD Procedure

Figure 56 presents the receive SRD procedure. The frame is discarded either if it is an invalid frame (with bit errors) or if it is not addressed to it (a master or a slave). Otherwise, if it is a valid frame addressed to the station, then it tries to find a match with one pre-configured message stream. If a match is found, then it builds a response frame using the value of the internal variable. After waiting T_{SDR}, it transmits the response frame.

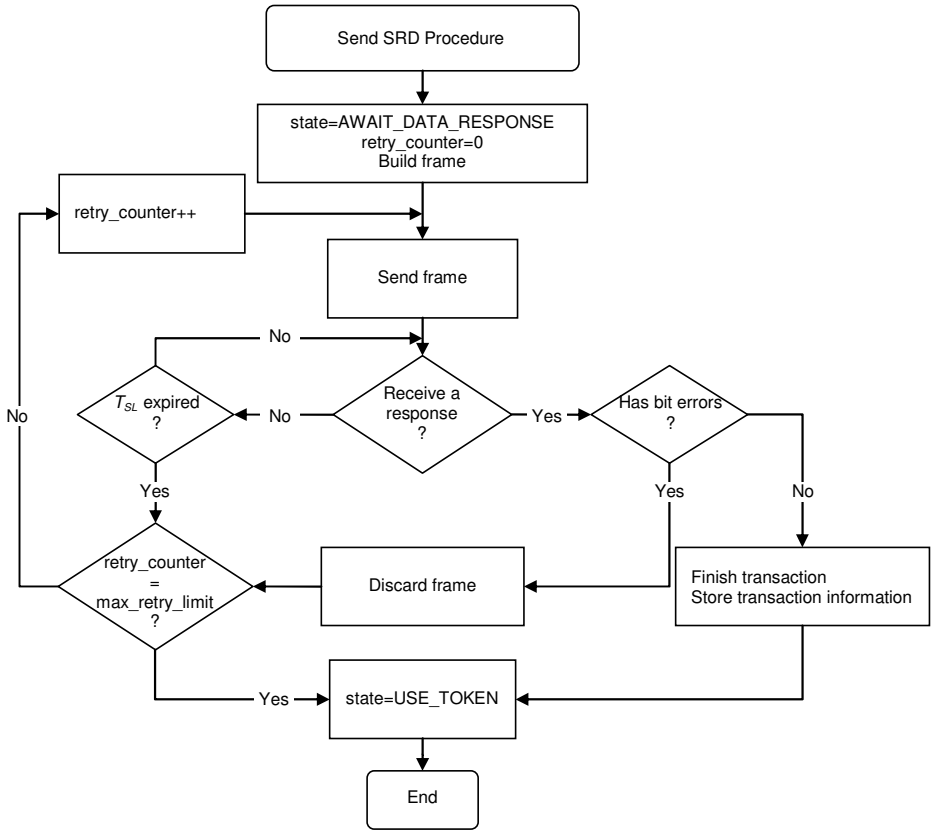

Figure 55 – Send SRD Procedure

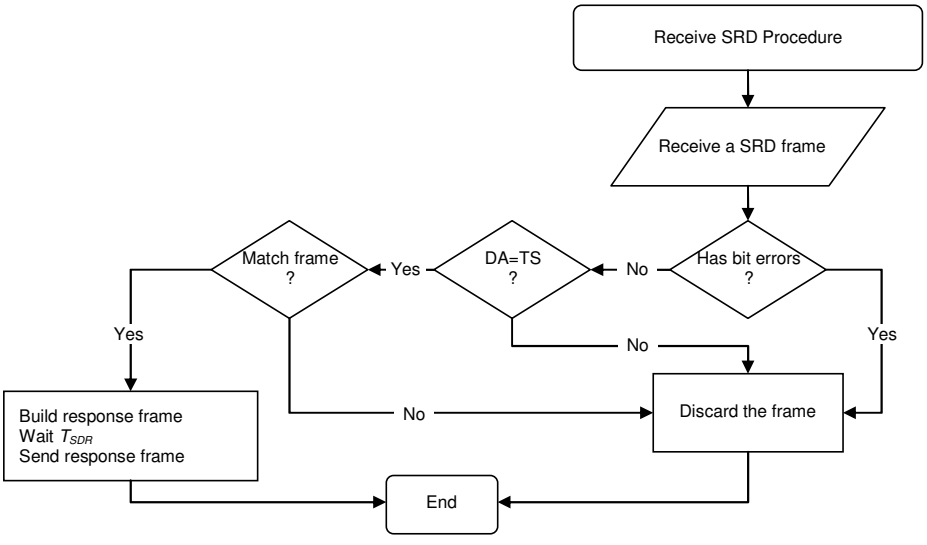

Figure 56 – Receive SDR Procedure

5.3. Repeater-Based Simulation Model Implementation

This simulator encompasses the functions concerning the PROFIBUS (described in Section 5.2), related to the domains interconnection and to the mobility procedure.

As previously mentioned, the repeater model is made up of two simple modules, `ComFunc` and `Connection_Point`. In our simulator implementation IS and BS functions are supported by the modules that model a repeater. It is assumed that the operation mode of the repeater is cut-through.

5.3.1. Interconnection

To interconnect different domains it is necessary to convert the received frame to the format of the destination domain and transmit the frame with a precise timing which guarantees minimal delays. For that purpose, there is the need to know the length of the DLL frame, the `Baud_rate` parameter value of the interconnected domains and how each frame is coded.

Figure 57 and Figure 58 illustrate a transaction between a `Master` module instance named M2 and a `Slave` module instance named S6, according to the network configuration presented in Figure 3, where M2 belongs to domain D4 and S6 belongs to domain D3.

In order to simulate bit by bit reception, `Connection_Point` module instance named CP7 delays the frame just enough time for it to know its length. Note that, the number of bits necessary to know the frame length depends on the PhL frame format and the PROFIBUS DLL frame type. For instance, wireless PhL frames include a head, tail fields and each DLL character is coded using 8 bits. Wired PhL frames do not include any head or tail fields, but each DLL character is coded using 11 bits. Concerning the frame length, PROFIBUS DLL has two kinds of frames: fixed and variable length frame. In order to know the length, in the first case, the first DLL character (SD field) is enough, but in the second case there is the need to receive the second DLL character, the LE field.

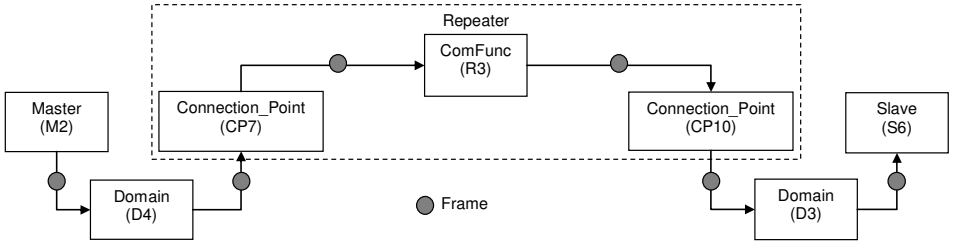

Figure 57 – Wired/wireless interconnection example

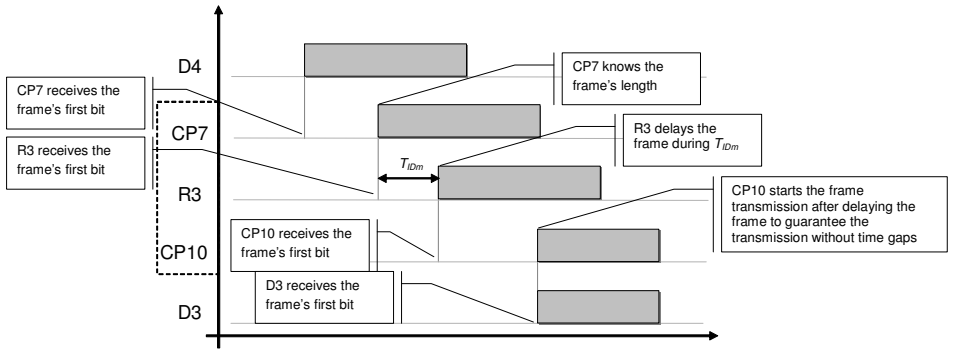

Figure 58 – Simplified timing behaviour of the module instances that model a repeater

After delaying the frame, CP7 passes the frame to the `ComFunc` module instance named R3. The frame is delayed by R3 during the time defined by a PDF configured by its parameters using the `_pdf_delay` prefix, after that, it passes the frame to the `Connection_Point` module instance named CP10. CP10 computes the frame last bit reception instant and the frame last bit transmission instant in

order to determine the first bit transmission frame instant in its domain. In this calculation the CP10 has to respect the idle time (defined by parameters that use `_pdf_tidm` prefix) between two consecutive transmissions.

The frame is transmitted to the `Domain` module instance named D3 by CP10 through `repeater_con` connection. D3 delays the frame according to the frame length and then the frame is transmitted through `domain_con` connections arriving at S6.

All these delays had been implemented according to the principles presented in [1] which guarantees that there is no need to queue a frame in the first repeater which relays a frame. Nevertheless, in the following repeaters the queuing of frames might occur. For that reason, each `Connection_Point` module instance is provided with a queue.

5.3.2. Mobility Procedure

The mobility procedure is triggered in a periodical fashion. The periodicity is defined by the `_tmob` parameter. In this implementation we assume that the MM is a dedicated master, i.e., it only controls the mobility procedure. For that reason, a mobility-related timer is loaded, with a value defined by the `_tmob` parameter, which is started when it starts operating.

After the mobility-related timer expires and when it holds the token frame, it broadcasts a `Beacon Trigger` (BT) frame. The BT is an unacknowledged frame, therefore after sending the BT frame it waits T_{ID2} to schedule the next action according to the message dispatching procedure presented in Section 5.2.3. Usually, it passes the token frame to its `Next Station` (NS). After sending the BT the mobility-related timer is reloaded.

`Connection_Point` module instances that are operating as BSs receive a BT frame and after relaying the received BT frame they start sending a pre-defined number (defined by `n_beacon` parameter) of `Beacon` frames (see Section 5.3.3 for more details). The BS waits for T_{IDm} before transmitting a `Beacon` frame.

Wireless mobile stations receive these `Beacon` frames and changes to domain according to the `_location_vector` parameter, which defines the domain location for each wireless mobile station (more details in Section 5.3.4).

5.3.3. Send Beacon Procedure

Figure 59 presents the send Beacon procedure. The first step is to set the `n_beacon_counter` variable equal to `n_beacon`, a BS parameter. After that, while `n_beacon_counter` variable is higher than zero, the BS waits T_{IDm}, builds a `Beacon` frame and then transmits the `Beacon` frame.

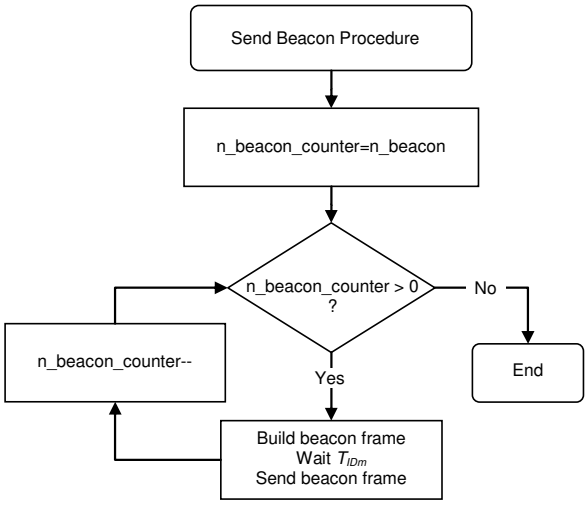

Figure 59 – Send Beacon Procedure

5.3.4. Stations Mobility

In order to model the mobility of a station between domains, in the Repeater-Based Hybrid Wired/Wireless PROFIBUS Network Simulator (RHW2PNetSim), the `Connection_Point`, which is operating as a BS, at reception of the `Beacon Trigger` frame from the `Mobility Master` (MM), sends to the `Controller` (through `ctrl_con` connection) a message indicating that `Beacon` frames will be transmitted. After ending the transmission of the `Beacon` frames, according to the procedure described in Section 5.3.3, they sends to the `Controller` a new message indicating that they finish the `Beacon` frames transmission.

The `Master` and `Slave` module instances which model wireless mobile stations, at reception of the `Beacon` frames, send to `Controller` (through `ctrl_con` connection) a message indicating which domain they want to change, according to their `_location_vector` parameter. The `Controller` manages this information in order to disconnect the `Master` or the `Slave` from `Domain` to which they are connected, and connect to the destination `Domain`. However, a `Master` or a `Slave` can only change to a domain where the `Beacon` frames were transmitted.

References

[1] M. Alves, "Real-Time Communications over Hybrid Wired/Wireless PROFIBUS-Based Networks", PhD. Porto: University of Porto, 2003.

[2] CENELEC, "General Purpose Field Communication System", vol. P-NET, PROFIBUS,WorldFIP: European Norm., 1996.

[3] J. Carvalho, A. Carvalho, and P. Portugal, "Assessment of PROFIBUS Networks Using a Fault Injection Framework". In proceedings of, 2005.

[4] A. Willig and A. Wolisz, "Ring Stability of the PROFIBUS Token-Passing Protocol Over Error-Prone Links", *IEEE Transactions on Industrial Electronics*, vol. 48, pp. 1025-1033, 2001.

[5] IEC, "IEC61158 - Fieldbus Standard for use in Industrial Systems": European Norm., 2000.

[6] M. Alves, et al., "Real-Time Communications over Hybrid Wired/Wireless PROFIBUS-based Networks". In proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, pp. 142-150, 2002.

[7] M. Alves, et al., "General System Architecture of the RFieldbus ": Deliverable D1.3. RFieldbus project IST-1999-11316, 1999.

[8] A. Varga, "OMNeT++ Discrete Event Simulation System," v2.3 ed, 2004. Available online at http:// www.omnetpp.org.
[9] P. Sousa and L. Ferreira, "Probability Distribution Functions " Polytechnic Institute of Porto, Porto, Technical-Report HURRAY-TR-070603, June 2007.
[10] A. M. Law and W. D. Kelton, "Simulation Modeling and Analysis", 3rd ed. New York: McGraw-Hill, 2000.
[11] P. Sousa and L. Ferreira, "Tools for Simulation Output Analysis " Polytechnic Institute of Porto, Porto, Technical-Report HURRAY-TR-070604, June 2007.
[12] P. Sousa and L. Ferreira, "Bit Error Models," Polytechnic Institute of Porto, Porto, Technical-Report HURRAY-TR-070602, June 2007.
[13] A. Burns and A. Wellings, "Real-Time Systems and Programming Languages Ada 95, Real-Time Java and Real-Time POSIX ", Third Edition ed: Addison Wesley Longmain 2001
[14] T. S. Rappaport, "Wireless Communications. Principles and Practice": Prentice Hall, 1996.
[15] P. Sousa and L. Ferreira, "Mobility Simulator," Polytechnic Institute of Porto, Porto, Technical-Report Hurray-tr-060403, April 2006.