1. RATE-ADAPTIVE TIME SYNCHRONIZATION (RATS)

1.1. Introduction

In the previous section it was shown that an instant time synchronization protocol can provide clock synchronization with an accuracy of a few microseconds. However, the problem with this approach is that nodes are quickly desynchronized and, therefore, in the long term, there is a big number of packets that is needed, in order to keep them synchronized with a specific time boundary. Especially in applications, where time accuracy is a major concern, such as synchronized sampling, data logging, coordinated actuation, etc, this approach does not constitute the optimal solution. The current state-of- the-art in this area are long-term time synchronization protocols. This section focuses on such a protocol, which is called Rate-Adaptive Time Synchronization (RATS).

RATS is a protocol that helps one node (child node) keep track of the local time of another node (parent node) in the network. In order to achieve this, the child node needs to store an array of tuples of the form $<T_{child}, T_{parent}>$. After the array is full, the child can use Least Square Means, in order to calculate the coefficients α, β of the linear equation $T_{PARENT\text{-}CURRENT} = \alpha + \beta * T_{CHILD\text{-}CURRENT}$, where $T_{CHILD\text{-}CURRENT}$ is the child's current time and $T_{PARENT\text{-}CURRENT}$ is the corresponding estimated time of the parent.

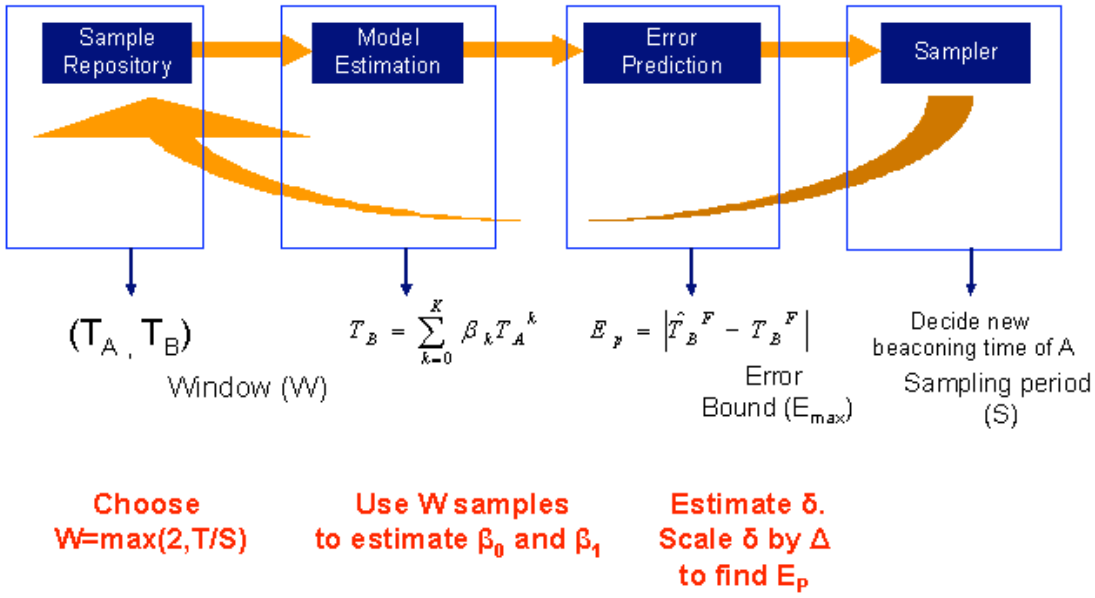

Figure 1.1 RATS structure

1.2. Protocol description

This section focuses on the description of RATS' components (Figure 1.1). The first step of the RATS algorithm (Figure 1.2) is to create a sample repository of tuples of the form $<T_A, T_B>$ that correspond to the timestamping pairs between nodes A and B. The optimal number of samples is calculated according to the current sampling period and the time constant T that depends on the environment. The equation that shows the size of the buffer is:

$$W = \max\{2, \frac{T}{Sampling\_Period}\}$$

The next step is the establishment of a model between the timestamping pairs. The chosen mathematical model is called Least Mean Square (LMS). This model is used to find the coeffiecients that relate 2 discrete signals. In our case, it has been proven that the best estimator is a linear one, therefore the equation will be of the form

$T_A = \alpha + \beta * T_B$, where α, β are shown by the following equations:

$$\alpha(offset) = \frac{\sum TS_{parent} * \sum (TS_{local})^2 - \sum TS_{local} * \sum (TS_{local} * TS_{parent})}{WindowSize * \sum (TS_{local})^2 - (\sum TS_{local})^2}$$

$$\beta(slope) = \frac{WindowSize * \sum (TS_{local} * TS_{parent}) - \sum TS_{local} * \sum TS_{parent}}{WindowSize * \sum (TS_{local})^2 - (\sum TS_{local})^2}$$

Afterwards, we need to calculate the error between the mathematical value and the real one. This is given from the equation

$predicted\ error = s\sqrt{1 + \frac{1}{n} + \frac{(x^* - \bar{x})^2}{\sum (x_i - \bar{x})^2}}$, where x^* is the estimated next local sampling time.

The final step is to compare the calculated error with an upper and a lower bound. If the error is larger than the upper bound, then the sampling period is increased, whereas if it is smaller than the lower bound, then the sampling period is decreased. RATS uses a Multiplicative increase, multiplicative decrease (MIMD) algorithm for the period changes, because it provides a quick convergence without big additional overhead.

```
Procedure calculate_new_period()
1. Compute W = max(2, T/period)
2. Find a, b from linear estimator.
3. Compute an estimated error bound
4. apply MIMD strategy
  if (Est Error < Lower Error Bound)
    Next_Period = period * 2
  Else if (Est Error > Upper Error Bound)
    Next_Period = period / 2
```

Figure 1.2 RATS algorithm

1.3. Implementation

In the beginning, the application that wants to establish time synchronization with another node sends a MSG_RATS_CLIENT_START message to the RATS module using post_short. The format of the post_short arguments are the following:

Data type	post_short parameter	Field name	Description
unsigned short	word	Node id	Id of the node, with whom time synchronization is requested
unsigned char	byte	Synchronization precision	Needed synchronization precision of the requested time conversion

Table 1.1 Format of rats_init_t struct

RATS adds this data to a list. This way it is possible for an application to do time synchronization with many other nodes or to have many applications do time synchronization with the same node, etc. Afterwards, RATS sends a MSG_RATS_SERVER_START message to the parent. At that moment both the parent and the child are in a learning state, which lasts until the child's buffer is full. Only after the buffer is full, will the child be able to calculate the coefficients that will be used to estimate the parent's time. In order to achieve that, the parent transmits BUFFER_SIZE (currently set to 4) packets of type MSG_TIMESTAMP to the child with an inter-packet period that is equal to INITIAL_TRANSMISSION_PERIOD (by default set to 2 seconds).

After the learning state is over, the children are able to use a linear estimator, which returns the values for α, β and the corresponding mathematical error that shows an estimation of the difference between the calculated value and the actual value of the parent's time. Afterwards, each child compares the value of the error to the value of the allowed estimation error that is needed by the application. If the estimation error is smaller than LOWER_THRESHOLD*allowed_estimation_error (LOWER_ THRESHOLD is currently set to 0.7), then the child sends a packet of MSG_PERIOD_CHANGE to the parent telling him to double the inter-packet period. If the error is higher than HIGHER_THRESHOLD*allowed_estimation_error (HIGHER_ THRESHOLD is currently set to 0.9), then the

MSG_PERIOD_CHANGE packet tells the parent to divide the current inter-packet period by 2.

At any point after the end of the learning period, the application is able to use RATS, in order to perform time conversions between the local time and the time of the other node. In order to do that, it needs to send RATS a MSG_RATS_GET_TIME message and pass as data a rats_t struct, which has the following format:

Data type	Field name	Description
unsigned char	mod_id	Id of module that requested the conversion
unsigned short	source_node_id	Id of node, whose time is the source of the conversion
unsigned short	target_node_id	Id of node, whose time is the target of the conversion
unsigned int	time_at_source_node	Time at source node
unsigned int	time_at_target_node	Time at target node (filled by RATS)
unsigned int	error	Error of conversion in milliseconds (filled by RATS)
unsigned char	msg_type	Return message type

Table 1.2 Format of rats_t struct

According to this scenario, it is possible that one parent might have many children, which might be using different inter-packet periods. In order to simplify the protocol, the parent transmits MSG_TIMESTAMP packets using the minimum interpacket period that was received.

Finally, if an application wants to terminate the time synchronization procedure with a node, it can send a MSG_RATS_CLIENT_STOP message to the RATS module, which sends a MSG_RATS_SERVER_STOP message to the parent. The MSG_RATS_CLIENT_STOP is sent using post_short and the format of the arguments is the same one as with MSG_RATS_CLIENT_START.

The sequence of events that was described above is shown in Figure 1.3.

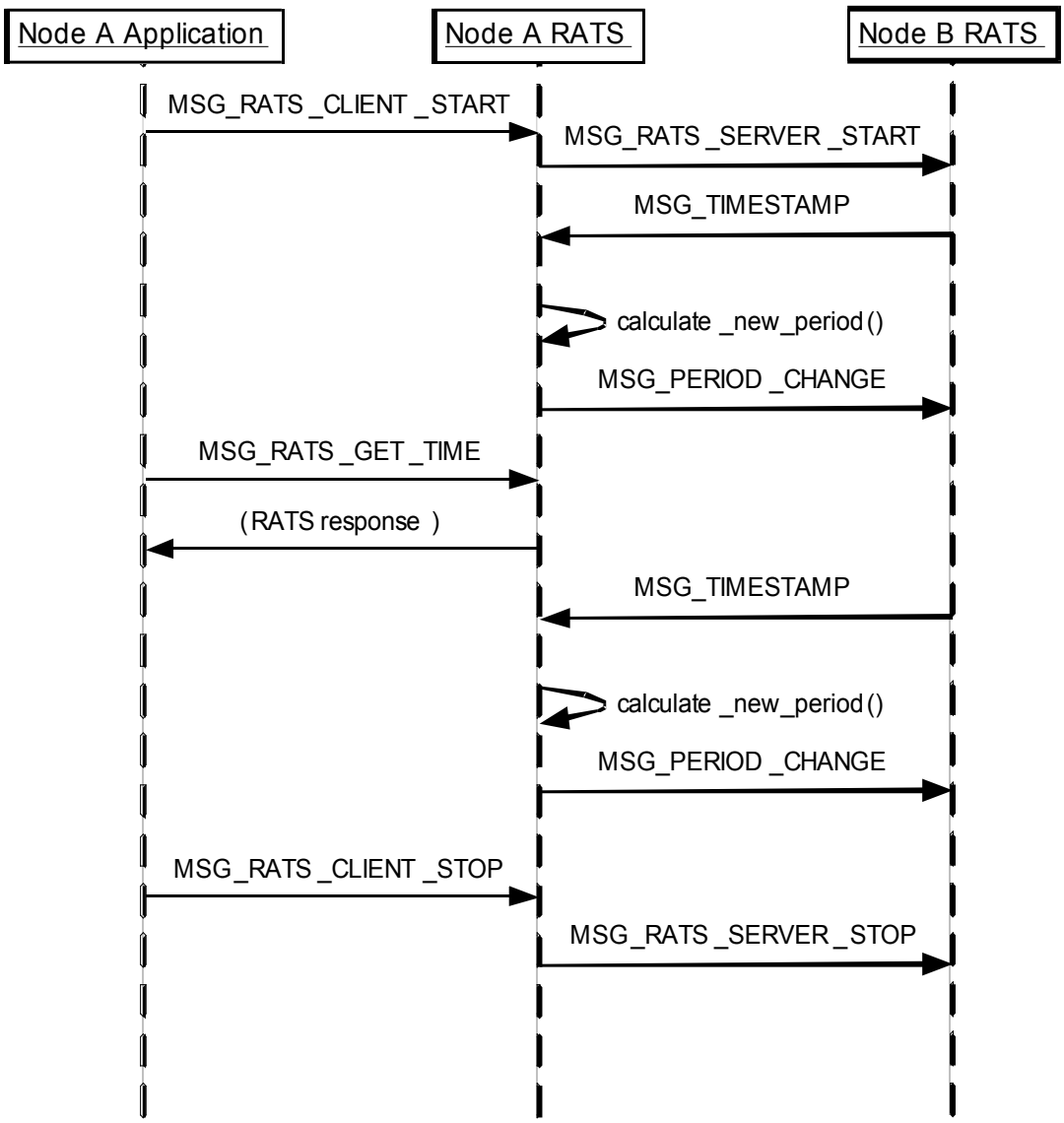

Figure 1.3 RATS packet exchange

1.4. Clock wrap-around

As it was noted in section 2, the maximum value for the timer that is used by the SOS timestamping library is 0x7F000000, therefore since its resolution is 1 clock tick every 8.6 microseconds, the timer will overflow after approximately 5 hours. After that, the clock will reset and start from 0. This wrap-around can cause problems to the linear estimator, which assumes that the input is monotonically increasing, since it is possible that some timestamps might have been collected before the wrap-

around and some others afterwards. In order to solve this problem, we need to detect the timer overflows and project the timestamps that occurred after the wrap-around to a reference clock. This can be easily seen in Figure 1.4, in which the last sample is projected, as if the wrap-around had not occurred.

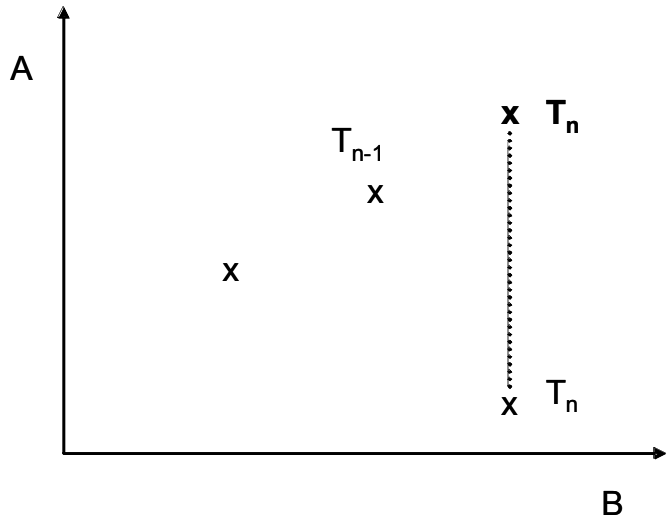

Figure 1.4 Timer wrap-around - clock projection

In this figure, each sample corresponds to a single T_i. In order to detect an overflow, we need to make sure that:

- For all the samples in the buffer we have $T_i < T_{i+1}$
- If the buffer holds N samples, then $T_N < T_{new\_sample}$

If one of the above cases is false, then there is an overflow and we need to add 0x7F000000 to the value of the new sample, before we pass the value to the linear estimator.

1.5. Packet formats

The MSG_TIMESTAMP packet has the following fields:

Data type	Field name	Description
unsigned int	time[0]	Time at the parent (filled by the network stack at the parent)
unsigned int	time[1]	Time at the child (filled by the network stack at the child)
unsigned char	type	Type (used to differentiate between normal packets that are used by the linear estimator and test packets, which are used, in order to verify the correct functionality of the protocol)
unsigned short	transmission_period	Inter-packet transmission period
unsigned char	min_period_node_id	Id of the node, which has set the transmission period

Table 1.3 Format of RATS MSG_TIMESTAMP packets

As it was described in section 2, all the packets of type MSG_TIMESTAMP are special for SOS. More specifically, after the Preamble and the Sync bytes of the packet have been transmitted, SOS timestamps the packet with the current time. This value is written in the first 4 bytes of the payload of the packet. On the other hand, when a MSG_TIMESTAMP packet is received, the current time is written at the second 4 bytes of the data payload. The application is responsible to allocate size for the payload that needs to be >=8 bytes.

In order to be able to timestamp packets that can be used in many different ways, the application can use the "type" field of the packet. In RATS, this field is used, in order to differentiate between the packets that are used for the calculation of the coefficients (type = NORMAL_PACKET) and the ones that are used, in order to validate the correct functionality of the protocol (type = TEST_PACKET). The way that this is verified will be analyzed below. The packets with type equal to TEST_PACKET have only the first 3 fields that were described above, since the verification test does not need the transmission_period and the min_period_node_id.

Finally, the packet that is sent by the child, in order to notify the parent that the period needs to change has the following 2 fields:

Data type	Field name	Description
unsigned short	old_period	Previous inter-packet period
unsigned short	new_period	New inter-packet period

Table 1.4 Format of RATS MSG_PERIOD_CHANGE packets

1.6. Fail-safe mechanisms

This section contains some design decisions that were taken, in order to make the protocol more robust, and explains the need for the previous packet formats. The scenarios that needed to be taken into consideration during the implementation of the protocol were:

a) How will we be able to minimize the packet transmissions?

b) What happens, if a packet from a child to its parent is lost or if the child "dies"? This scenario affects the case, where the child into consideration is the one that has set the minimum inter-packet transmission period of the parent.

c) What happens, if a packet from the parent to its children is lost or if the parent "dies"?

Regarding the first scenario, we needed to take into consideration the fact that the parent actually uses only the smallest period that is sent to him and discards the rest of the values. This means, that if all the children transmit a packet to the parent each time that they calculate a new period, then the parent will discard the vast majority of these packets without any further processing. In order to avoid that, we have changed the protocol so that a child sends a packet only if the newly-calculated period is smaller than the one used by the parent or if the current node is the one that has set the minimum period of the parent (this is the reason that the packets sent by the parent include both the transmission_period and the min_period_node_id). In

order to make the protocol more robust and to avoid any inconsistencies, the parent also conducts the same checks, when it receives a MSG_PERIOD_CHANGE packet.

This optimization reduces the number of transmissions a lot, however now we need to take care of the fact that the node, which has set the minimum period, might eventually “die”. In this case, the parent would be needlessly transmitting packets very often, even though the rest of the “living” nodes of the network might not need them. In order to take care of this scenario, we have added another type of packet, which is named MSG_VALIDATE. More specifically, the parent sends this type of packet, if transmission_period+4*INITIAL_TRANSMISSION_PERIOD time has passed and it has not received any packet that is either sent from the child that has set the minimum id or contains a period that is smaller than the currently used one. The parent retransmits the same packet for RECOVERY_RETRANSMISSIONS times (set to 5 by default) with a frequency of 1 packet every INITIAL_TRANSMISSION_PERIOD seconds. If the parent receives a response before RECOVERY_RETRANSMISSIONS have been made, then the protocol continues normally. Otherwise, the parent broadcasts a MSG_TRANSMIT_PERIODS packet, asking for the periods of all the nodes. This packet is retransmitted twice (once every INITIAL_TRANSMISSION_PERIOD seconds). After that period the parent uses the lowest period that has been transmitted by the children as the new transmission period. If the parent doesn’t receive any replies, then the execution of the protocol is stopped, because the parent considers that there are no children, which wish to synchronize with him. Another fail-safe mechanism for this scenario is that if a child receives a MSG_TIMESTAMP packet from the parent and the transmission_period is bigger than the transmission period that is required by the child, then the child immediately sends a packet to the parent

telling him to use a smaller period. This way we manage to eliminate the ACKs (since eventually the parent will receive a packet from the child that has the minimum period, even if certain packets get lost).

The final scenario that needs to be taken into consideration is what happens, if the child doesn't receive a MSG_TIMESTAMP packet from the parent, after the inter-packet transmission period has elapsed. In this case, the child transmits a MSG_PANIC packet to the parent, who needs to respond immediately. If the child doesn't receive any response, then it sends the same packet for PANIC_RETRANSMISSIONS times (currently set to 5) with a rate of 1 packet every INITIAL_TRANSMISSION_PERIOD seconds. If after this period, there is no reply, then the parent is considered "dead" and the child stops trying to synchronize with the parent.

1.7. Protocol verification

In order to verify the correct execution of the protocol we use an external node, which broadcasts a MSG_TIMESTAMP packet with "type" equal to TEST_PACKET. The parent forwards the packet to the serial port. Another program, which is running on the computer, receives the packet and prints the time that the packet was transmitted by the parent and the time that the packet was received by the child. When a child receives the above packet, it sends another packet to the parent, which includes the time that the time was sent by the external node and the estimation of the parent's current time. The parent forwards the packet to the serial port, where it is received by the application. Since the time that the parent and the child received the external packet should be approximately the same, we can see how close the

estimation of the parent's time (that was calculated by the child) and the actual parent's time are.