Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs

Jan Romberg
romberg@in.tum.de

Andreas Bauer
baueran@in.tum.de

Technische Universität München, Institut für Informatik
Boltzmannstrasse 3, D-85748 Garching, Germany

ABSTRACT

Dataflow synchronous languages have attracted considerable interest in domains such as real-time control and hardware design. The potential benefits are promising: Discrete-time semantics and deterministic concurrency reduce the state-space of parallel designs, and the engineer's intuition of uniformly progressing physical time is clearly reflected. However, for deriving implementations, use of synchronous programs is currently limited to hardware synthesis, generation of non-distributed software, or deployment on time-triggered architectures.

In this paper, it is shown how synchronous dataflow designs can be used for synthesizing distributed applications based on target architectures that do not provide a global time base by default. We propose a distribution method called "synchronization cascade" where the nodes' local clocks depend on each other in a tree-like manner. For evaluation of the method, we characterize some requirements for firm real-time applications, and evaluate our approach with respect to the postulated requirements.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Languages

Keywords

Embedded Software, Synchronous Languages, AutoFocus, Code Distribution, Scheduling

1. INTRODUCTION

Dataflow synchronous programs and specifications, exemplified by LUSTRE and SIGNAL[2], or AUTOFOCUS[7], provide a discrete-time abstraction from real-time, concurrent implementations. This abstraction is familiar to control engineers and hardware designers: all parts of the program run in a uniform timeframe (time-synchronous). Concurrency in synchronous designs is based on lock-step composition, which usually results in more understandable designs, and avoids a blow-up of the state space of parallel programs.

While synchronous programs are well suited for synthesis of monolithic hardware and software, the problem of *distributed* implementation of a synchronous program is still a challenging one. Clearly, formal abstractions of distributed systems, along with an automated procedure to synthesize distributed implementations, are desirable for several reasons, including improved behavioral validation, for instance by simulation or formal verification.

However, there are numerous challenges to an (preferably automated) approach for distribution of synchronous programs, e. g. (1) integration of existing systems with the synthesized executives, (2) required compatibility with the numerous platforms in the domain, (3) compliance with tight nonfunctional requirements such as timeliness, memory consumption, and hardware cost.

Looking at real-time control application in the automotive sector, time-triggered protocols [8] are an attractive target platform for highly critical applications such as X-By-Wire. However, for applications where cost concerns and legacy integration issues are more dominant compared to criticality requirements, existing *event-triggered* bus architectures such as Controller Area Network (CAN) may play an important role for some time to come. Applications characterized by this requirement will be called *medium-criticality applications* in the sequel.

Synchronous approach vs. firm real time

In the context of medium-criticality applications, let us discuss the synchronous distribution issue in more depth: Certainly, semantically correct implementation of the distributed program is vital. But when looking at the state of the art in distributed real-time control applications, many of these applications meet their timing and criticality requirements even though they are based on communication media that provide no absolute guarantees about response times. As a possible explanation, some control applications are known to tolerate the loss of a bounded number of messages, e. g. state values. In real-time systems, this corresponds to the notion of *firm real-time*: transactions are discarded when they miss their deadlines, as there is no value to completing them afterwards. In contrast to hard real-time systems, a bounded number of deadline misses is not considered fatal. How can the notion of firm real-time be married with the distribution of synchronous programs?

It can also be questioned whether synchronous implementations require the existence of a precise global timebase. Existing works on asynchronous distribution of synchronous programs [5] have shown that this is not necessarily the case. On the other hand, it is quite clear that asynchronous distribution does not satisfy some requirements of real-time control applications when communica-

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

tion media are involved that allow message losses or unbounded latencies. Can we use a loose timebase, which may be cheaper to implement, and still obtain implementations suited for real-time control applications?

Based on this discussion, our approach is twofold: (1) Provide a distribution method for synchronous programs based on a synchronization / communication layer with a loose timebase. The method should ensure semantically correct execution of the synchronous program under normal operation conditions. (2) Make sure the synchronization / communication layer provides a reduced service, including synchronization, in case of certain faults. By adjusting some well-defined parameters, it is then up to the developer to ensure that the system remains in normal operation for the most part of its lifecycle, and meets the correctness and timeliness requirements imposed by the application.

In the following, we will describe the procedure to deploy synchronous programs onto event-triggered networks based on loose synchronization, and evaluate our method with respect to the requirements of medium-criticality applications. Section 2 introduces a simple denotational style [4] used for specifying synchronous programs and abstractions of communication channels. Section 3 introduces a synchronization / communication layer called *synchronization cascade*, which is used for distributed deployment of synchronous programs. Section 4 defines some requirements for distribution of synchronous programs for medium-criticality applications, and shows that our distribution method satisfies some essential properties related to these requirements. Section 5, finally, relates our approach to works of other authors, and gives an outlook discussing future directions.

2. DATAFLOW SYNCHRONOUS SPECIFICATIONS

We model distributed software as a network of *components* communicating over timed *streams*.

Streams

A stream is a finite or infinite sequence of *messages* from a set M. For such a set of messages, we use $M^\omega = M^* \cup M^\infty$ to denote the set of all finite and infinite streams over M. For a stream $\sigma \in M^\omega$, the i-th message is written as $\sigma.i$. We define a special message $\perp$, the *absent message*: for a given set M, we write $M_\perp = M \cup \{\perp\}$ for the set obtained by adding the absent message.

The *length operator* $\#$ yields the length of the stream to which it is applied. *Concatenation* of streams, written $\sigma_1 \& \sigma_2$, yields a stream that starts with the messages of σ_1 followed by the messages of σ_2. The *filtering operator* Ⓢ is used to filter away messages. $M' Ⓢ \sigma$ is the substream of σ obtained by removing all messages in σ that are not in the set $M' \subseteq M$.

In the following, we will use a *time-synchronous interpretation* of streams: for all streams, the position of a message in a stream is associated with a unique instant in a uniform discrete timeframe.

Components

A component c has a set of *input signals* $I = \{i_1, i_2, \ldots, i_m\}$ with types M_{i_j} and a set of *output signals* $O = \{o_1, o_2, \ldots, o_n\}$ with types M_{o_j}. $\mathcal{I} = M_{i_1}^\omega \times M_{i_2}^\omega \times \cdots M_{i_m}^\omega$ and $\mathcal{O} = M_{o_1}^\omega \times M_{o_2}^\omega \times \cdots M_{o_n}^\omega$ are the input and output domains of the component, respectively. We consider only *deterministic components* that are *complete on their inputs*: for these, inputs and outputs can be related by a total function $f : \mathcal{I} \to \mathcal{O}$. We call this function f the *I/O function* of the component. For time-synchronous streams, we require I/O functions to be *causal*; that is, present outputs do not depend on future inputs, and *length-preserving*, i.e. there is a fixed correspondance between the lengths of inputs and outputs.

Networks of Components

For composition of two components c_1 (I/O function f_1, inputs I_1, outputs O_1) and c_2 (I/O function f_2, inputs I_2, outputs O_2) yields a component with inputs $(I_1 \backslash O_2) \cup (I_2 \backslash O_1)$, and outputs $O_1 \cup O_2$. Semantically, composition of c_1 and c_2 is then defined by intersection of c_1's and c_2's behavior on the shared inputs/outputs (synchronous composition); we restrict the scope of our semantics to the class of systems where the intersection is again a (total) I/O function. Such networks of components can be specified in a graphical style, using rectangles for components, and directed arrows for signals/channels [4].

Abstractions for Communication Channels

In the following, we will use components for abstractly defining properties of a given communication channel. This definition will be used in Sections 3 and 4 to formalize assumptions about the communication medium and guarantees provided by the synchronization / communication layer.

DEFINITION 1 (CHANNEL). *A component c is a* channel *iff $|I| = |O| = 1$ and $\mathcal{I} = \mathcal{O}$.*

DEFINITION 2 (m-LENGTH-PRESERVING CHANNEL). *Given some $m \geq 0$, a channel is an* m-length-preserving channel *iff $\forall \sigma_I \in \mathcal{I}.\#(f(\sigma_I)) = \#(\sigma_I) + m$.*

DEFINITION 3 (UNIT DELAY CHANNEL). [1] *A channel is a* unit delay channel with initial message m *iff $\forall \sigma_I \in \mathcal{I}.f(\sigma_I) = m \& \sigma_I$.*

As a corollary, unit delay channels are 1-length-preserving.

DEFINITION 4 (n-BOUNDED LOSSY CHANNEL). *Given some $n > 0$ and some m-length-preserving channel ch with domain $\mathcal{I} = \mathcal{O} = M_\perp$, ch is an* n-bounded lossy channel *iff, for all input/output streams $(\sigma_I, \sigma_O) \in f$, for all of σ_I's substreams σ_I^i of length n at position i, for all of σ_O's substreams σ_O^{i+m} of length n at position $i + m$, the following condition holds:*

$$\#(M Ⓢ \sigma_I^i) = n \implies \#(M Ⓢ \sigma_O^{i+m}) \geq 1$$

Intuitively, if fed with messages from the set M, an n-bounded lossy channel will lose at most $n - 1$ subsequent messages. As a direct consequence of the above definition, any n-bounded lossy channel is also $n + 1$-bounded lossy.

3. SYNCHRONIZATION CASCADES

3.1 Terminology

A *synchronization cascade* provides a layer for synchronization and communication, and implements a logical network topology on top of some suitable physical topology where each link in the logical topology can be mapped to a physical counterpart. We call the underlying protocol(s) the *base protocol(s)* of the cascade.

A synchronization cascade is a rooted tree with nodes $\mathcal{N} = \{N_0, N_1, \ldots\}$, and edges $S \subseteq \mathcal{N} \times \mathcal{N}$. Each node corresponds to a processor or control unit in the distributed implementation. Edges

[1] Our unit delay definition corresponds with a combined use of the `pre` and `->` operators in LUSTRE.

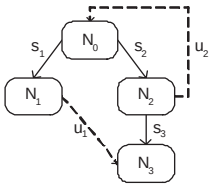


Figure 1: Example for a synchronization cascade

$s \in S$ are called (direct) *synchronizing links*: Each such link communicates a periodic message that is used by its child node to synchronize itself with the parent node.

The root of the tree is called *master node* N_0. For a non-master node N, we denote as $Li(N)$ the set of those synchronizing links that form a path from N_0 to node N. If $(N, N_0) \notin S$, the links in $Li(N)$ form an *indirect link* from N to N_0. $Par(N)$ is the set of *parent nodes* along the path such that $N_0 ß Par(N)$ and $N \notin Par(N)$.

The rooted tree is extended to a (directed) multigraph by adding edges $U = \{u_1, u_2, \dots\}$ (depicted as dashed edges). The edges $u \in U$ are called *nonsynchronizing links*: while their value is usually important to the receiver, the timing of their reception does not influence the receiver's activation times.

An example for a synchronization cascade is shown in Fig. 1: Node N_0 is the master node. Links s_1, s_2, s_3 are synchronizing links, while u_1 and u_2 are nonsynchronizing. The master node emits a periodic synchronizing message with a predefined *base period* T.

Processing phases

Fig. 2 shows, schematically, the timing of computations performed by a single node. During each cycle, the node performs two subsequent computations: a *send/receive* phase, and a *computation* phase. For given $N_i \in \mathcal{N}, j \in \mathbb{N}_0$, instant $t_{i,j}$ denotes the activation instant of node N_i at step j.

Send/receive phase

The send/receive phase is triggered by a periodically elapsing timer for the master, and by the respective synchronizing message for a non-master node. During this phase, the nonsynchronizing messages received since the last send/receive phase and the incoming synchronizing message are read, and all outgoing messages computed in the last computation phase are emitted. Because the send/receive phase requires nonzero time for execution, and the receiver node could potentially lose synchronizing messages if their interarrival time is too short, we define a *quiet interval* that overlaps the send/receive phase, and during which the node is not required to process incoming synchronizing messages. The remaining part of the cycle is called the *receptive interval*. For nonsynchronizing messages arriving in the quiet interval, the node may either read the message immediately, or leave it in the message buffer so it can be processed by the next send/receive phase. The quiet interval $(0, Q]$ starts at the beginning of each period. The analysis below will ensure that a node does not receive synchronizing messages in $(0, Q]$ under given operating conditions. Q is typically a worst-case estimate of the send/receive period's combined task response and execution times. In the following, we formally require $0 \leq Q < T \cdot (1 - 2\varepsilon)$, where ε is a clock drift constant introduced in Section 3.2. Because communication overhead is usually small compared to computation time, we expect typical assignments for Q to be less than $T/4$.

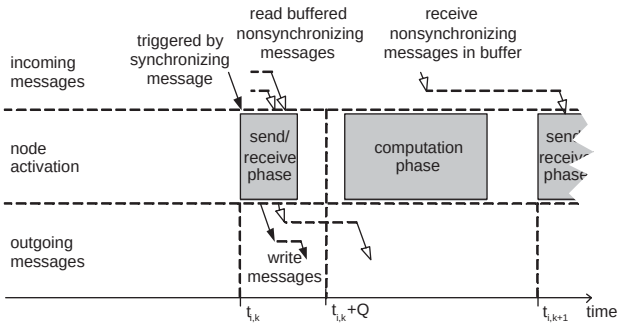


Figure 2: Processing phases for step k of a node N_i. Filled arrowheads denote synchronizing messages, empty arrowheads correspond to nonsynchronizing messages

Computation phase

During the computation phase, the local part of the distributed program is executed, the received messages are processed, and the next values of the outgoing messages are computed. Outgoing messages are buffered till the next send/receive phase.

Note that the computation phase may be interrupted by the next send/receive phase under certain circumstances. It is assumed that the send/receive handler uses default values for all of those outgoing synchronizing messages where no value has been computed in the last step. Consequently, the availability of a synchronization message for the next cycle does not depend on the completion of the computation phase.

Activation of the send/receive phase

Each node N_i defines the following functions and variables:

- getSynchronizingMessage: M yields the value of the current synchronizing message.
- sendReceive: $M \rightarrow \#, ff$ executes the processing phase given a synchronizing message, and yields a boolean value whether the execution was successful.
- getDefaultMessage: M yields a default message for the synchronizing message, e. g. based on the last available values of the message.
- state $\in$ {EXTERNALLY_TRIGGERED, MESSAGE_ABSENT, SELF_TRIGGERED} is a state variable.
- timer$_i \in \mathbb{R}$ is the physical timer of node N_i.
- count $\in \mathbb{N}$ is a counter.

The send/receive phase of each node is initiated by two tasks, message_available_task and timer_task. timer_task is activated T time units and, if necessary, T_{ma} time units after the last activation. (The meaning of T_{ma} will be explained in the next paragraph.) timer_task has an idealized release time of zero. The two tasks and the states and transitions of the activation algorithm are shown in Fig. 3.

States and transitions

After initialization of the cascade, the master node is in state SELF_TRIGGERED, all other nodes are in state EXTERNALLY_TRIGGERED. In state EXTERNALLY_TRIGGERED, the respective node is synchronized with its parent node, and the send/receive phase is

```
message_available_task:
  if state ∈ {EXTERNALLY_TRIGGERED,
        MESSAGE_ABSENT} then
    timer_i := 0
    state := EXTERNALLY_TRIGGERED
    sendReceive(getSynchronizingMessage()) endif
timer_task:
  if state = SELF_TRIGGERED then
    if timer_i = T then
      timer_i := 0
      sendReceive(getDefaultMessage()) endif
  else if state = MESSAGE_ABSENT then
    if timer_i = T then
      timer_i := 0, count := count + 1
      if count ≥ n_pf then
        state := SELF_TRIGGERED endif
      sendReceive(getDefaultMessage()) endif
  else
    if timer_i = T_ma then
      timer_i := 0, count := 0
      state := MESSAGE_ABSENT endif endif
```

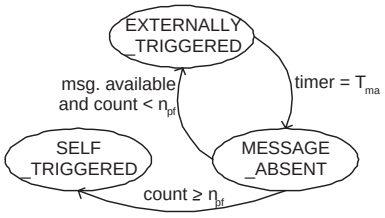


Figure 3: Activation, states, and transitions of a node N_i

periodically activated by the synchronizing message. In state MESSAGE_ABSENT, the node has detected a (possibly transient) absence of the synchronizing message. The send/receive phase is activated by the node's own periodic timer in this state. We will show in Section 3.4 that, while in state MESSAGE_ABSENT, the node is able to re-synchronize itself with its parent node. In state SELF_TRIGGERED, the node is periodically activated by its own timer, and there are no guarantees about the node's ability to re-synchronize itself with its parent node, if existent. The parameter T_{ma} is called the *message absence detection margin*. It denotes the time interval after which, if no synchronizing message has been detected, a node in state EXTERNALLY_TRIGGERED changes to MESSAGE_ABSENT. Parameter n_{pf} is the *parent fault detection count*. It denotes the maximum number of periods the node will remain in state MESSAGE_ABSENT if no synchronizing message is detected. If this number exceeds n_{pf}, the node will change to state SELF_TRIGGERED. Sender fault detection therefore initiates a fallback behavior in case either the parent node or the communication medium fails for a longer period of time. Note that re-synchronization with the parent after the node has entered state SELF_TRIGGERED is not in the scope of this paper.

3.2 Environment assumptions

We will now state some assumptions about the physical environment of the cascade. The assumptions will be necessary in order to show that the cascade meets its operational requirements. Some of the assumptions will be required independent of the network state, while others are prerequisites for normal operation of the network, and may be violated under fault conditions.

Physical clocks

Each node N_i has its own physical clock timer_i. A physical clock is typically subject to drifts and jitter w.r.t. the ideal physical time t. Operation of a synchronization cascade requires that deviations of all the nodes' clocks from ideal time are bounded by a constant.

DEFINITION 5 (ε-BOUNDED CLOCK DRIFT). *For a given cascade, let each node N_i be associated with a physical clock* timer_i. *The cascade is said to have an* ε-bounded clock drift *iff, for all intervals where* timer_i *is not reset,*

$$\forall N_i \in \mathcal{N}. \left| \frac{d\mathsf{timer}_i}{dt} - 1 \right| \leq \varepsilon$$

In combination with our definition of timer_task, the bounded clock drift assumption guarantees that the physical base period of each node is bounded by $[T/(1+\varepsilon), T/(1-\varepsilon)]$, and the message absence detection period is bounded by $[T_{ma}/(1+\varepsilon), T_{ma}/(1-\varepsilon)]$.

Message jitter

We define for each link s_j, u_j in the cascade a map Δ_{li} mapping direct links, i. e. links between adjacent nodes, to their corresponding worst case *message jitters*, assuming that some adequate method for analysis is available[2]. The minimum and maximum message latencies for direct or indirect links $(N_i, N_{i'})$ will be denoted as $d_{min}(i,i'), d_{max}(i,i')$, respectively, such that

$$d_{max}(i,i') - d_{min}(i,i') = \textstyle\sum_{s_j \in Li(N)} \Delta_{li}(s_j) \qquad (1)$$

holds for all (N, N').

The message jitter summarizes the end-to-end jitter from the instant the send/receive phase at the parent is activated until the child node's activation time. The worst-case jitter will typically include (1) execution time jitter of the sender's send/receive code, (2) queuing jitter at the sender, (3) communication jitter of the medium, (4) response time jitter of the receiver's task.

Because the message jitter includes the queuing jitter at the sender, the bound may be invalid if the communication medium is not accepting messages (e. g. due to unforeseen overload conditions or external disturbances). We therefore assume the existence of a simple *communication layer* that enforces the predetermined queuing interval by retracting the message when the precomputed worst-case queuing time is overrun. Note that this typically requires the layer to have some access to lower-layer operations of the controller.[3]

For correct operation of the cascade, the end-to-end jitter from the master to any node must be bounded:

DEFINITION 6 (BOUNDED SYNC. MESSAGE JITTER). *Let $Li(N)$ denote the set of all synchronizing links that form a path from the master to the node N. The network is said to have a* bounded synchronizing message jitter *iff*

$$\forall N \in \mathcal{N}. \textstyle\sum_{s_j \in Li(N)} \Delta_{li}(s_j) < \min\left(\frac{T-Q}{2}, \frac{T(1-5\varepsilon)}{2}\right)$$

This bound should be satisfiable for a large number of practical applications. For instance, in an automotive case study described in

[2]For the CAN protocol, the analysis described in [9] yields both bounds for worst-case response times and message jitters on the bus.

[3]In the case of the CAN protocol, the two most popular controller ICs (Intel 82527 and Philips 82C200) allow to retract messages after they have been put in the send buffer

[9], for the case of a 1MBit/s CAN bus, most high-priority messages have a jitter of around $10^{-3}s$, so for $\varepsilon = 10^{-6}$, $T = 10^{-2}s$, and $Q = T/20$, cascades up to depth 4 (four synchronizing links between master and the "farthest" node) are possible.

Message loss

The synchronization mechanism has to meet certain fault tolerance requirements. A typical fault in event-triggered real-time systems is the loss of a message: the loss can be caused by the sender when aborting a send operation (e. g. if the queuing delay is longer than expected, and a newer value is available), or by the communication medium itself. Seen more abstractly, we can associate message loss with the *input/output behavior of a link* in the cascade. The following definition will capture this:

DEFINITION 7 (I/O FUNCTION OF A (DIRECT) LINK). *For a given execution of a cascade, the* I/O function of a link $l \in S \cup U$, *written f_l, is defined as the function mapping the sequence of messages written by the sender's program to the sequence of messages arriving at the receiver node, where the special output symbol $\perp$ indicates a lost message.*

DEFINITION 8 (I/O FUNCTION OF AN INDIRECT LINK). *For a given execution of the cascade, the* I/O function of an indirect link $l = l_1 \rightarrow l_2 \rightarrow \cdots \rightarrow l_m$, *where $l_i \in S \cup U$ and l_1 and l_m are the first and last links in the direction of message flow, respectively, is defined as the composition of the individual links' I/O functions:*

$$f_l = f_{l_m} \circ \cdots \circ f_{l_2} \circ f_{l_1}$$

Using these definition, a *lossy* link models both message loss due to the sender's communication layer aborting the send, and due to the medium losing messages. In order to define *bounded* message losses, we will use the definition of n-bounded lossy channels from Section 2 to capture the condition that a cascade may suffer from a bounded number of message losses on each of its direct and indirect synchronizing links.

DEFINITION 9 (n-BOUNDED LOSSY CASCADE). *A cascade with master N_0 is an* n-bounded lossy cascade *iff, for all executions of the cascade, for all direct and indirect synchronizing links from N_0 to nodes $N \in \mathcal{N}$, the link's input/output function is an n-bounded lossy channel.*

3.3 Choice of Parameters

This section gives some predefined values for the parameters T_{ma} and n_{pf} used by the send/receive phase activation algorithm. T_{ma} is chosen such that loss of the synchronizing messages cause the receiver's activations to be delayed by $T/2$ w.r.t. the master's activation instants, while n_{pf} results from an analysis of the maximum number of lost messages that can be tolerated by the synchronization algorithm:

$$T_{ma} = \frac{3}{2}T, \tag{2}$$

$$n_{pf} = \max_{N \in \mathcal{N}} \left(\left\lfloor \frac{1}{4\,T\varepsilon} \left(T - \left(2 \sum_{s_j \in Li(N)} \Delta_{li}(s_j) + \max(2Q, T\varepsilon) \right) \right) \right\rfloor \right), \tag{3}$$

where the computed value for n_{pf} is required to be positive. For $\varepsilon \ll 1$, $Q \ll T$, the choice for T_{ma} can be shown to be very close to an optimally robust assignment, so that a maximal number of lost synchronizing messages can be tolerated in the presence of clock drifts. We will demonstrate in the next section that, for our assignment for T_{ma}, the cascade indeed satisfies this robustness requirement for an n_{pf}-bounded number of lost messages.

3.4 Analysis of Operational Modes

This section will provide an analysis of the different operational modes of the cascade: operation under *normal conditions*, operation under *transient fault conditions*, and operation under *permanent fault conditions*. For normal operation, we will show that all non-masters remain in state EXTERNALLY_TRIGGERED, while under transient fault conditions, it will be shown that non-masters never enter state SELF_TRIGGERED.

The following two definitions deal with the property of *synchronization* – the offset of the node's activation instant w.r.t. the master's activation instant is bounded – and *receptiveness* – an assertion about the principal ability of a node to receive the synchronizing message for the current step during its receptive interval. Note that for forwarding a synchronizing message over the entire length of an indirect link, *all* nodes along the link have to be receptive.

DEFINITION 10 (j-SYNCHRONIZATION). *For a node N_i and a step $j \in \mathbb{N}_0$, the statement "N_i is j-synchronized" corresponds to the property*

$$d_{\min}(0,i) \leq t_{i,j} - t_{0,j} \leq d_{\max}(0,i) \tag{4}$$

Nodes in $\mathcal{N}$ are assumed to be 0-synchronized (proper initialization of the cascade). Furthermore, we define that the master node N_0 is j-synchronized for all $j \in \mathbb{N}_0$. Again, j-synchronization is extended to sets of nodes and indices.

DEFINITION 11 (j-RECEPTIVENESS). *For a node N_i and a step $j \in \mathbb{N}$, the statement "N_i is j-receptive" corresponds to the three properties*

$$t_{0,j} + d_{min}(0,i) > t_{i,j-1} + Q, \tag{5}$$

$$(j-1)\text{−synchronized}(N_i) \implies t_{0,j} + d_{max}(0,i) < t_{i,j-1} + \frac{T_{ma}}{1+\varepsilon}, \tag{6}$$

$$\neg(j-1)\text{−synchronized}(N_i) \implies t_{0,j} + d_{max}(0,i) < t_{i,j-1} + \frac{T}{1+\varepsilon}, \tag{7}$$

where $T_{ma}/(1+\varepsilon)$ is the minimum message absence detection period defined in Section 3.2. The master node N_0 is defined to be j-receptive for all $j \in \mathbb{N}$. j-receptiveness is extended to sets of nodes $\mathcal{N}' \subseteq \mathcal{N}$ and to sets of indices $J \subseteq \mathbb{N}$ such that $\mathcal{N}'$ is j-receptive iff all of its members are, and N_i is J-receptive iff N_i is j'-receptive for all members $j'' \in J$.

DEFINITION 12 (NORMAL OPERATING CONDITIONS). *A cascade is said to be under* normal operating conditions *iff it is* 1*-bounded lossy, and the ε-bounded clock drift assumption holds.*

LEMMA 1. *Let N_i be a non-master node in a cascade. Then j-receptiveness of N_i and arrival of a synchronizing message in step j at N_i imply j-synchronization of N_i.*

PROOF. By Definition 11, it follows from j-receptiveness of N_i that, if a synchronization message originating from N_0 arrives at N_i for step j, it is received during the receptive interval of N_i, leading to an activation of N_i at $t_{i,j}$. According to our assumption about the communication medium, the difference $t_{i,j} - t_{0,j}$ is then bounded by $[d_{\min}(0,i), d_{\max}(0,i)]$, so j-synchronization holds for N_i. □

LEMMA 2. *Let N be a non-master node in a cascade under normal operating conditions. Then j-receptiveness of N and Par(N) imply j-synchronization of N.*

PROOF. Straightforward adaptation of Lemma 1: Normal operating conditions ensure that the direct or indirect synchronizing link to N is 1-bounded lossy, so synchronizing messages from N_0 are never lost. Because N_0 will send a message in every step j, and both N and N's parent nodes are j-receptive, a synchronizing message will be received by N in its receptive interval for all j. □

LEMMA 3. *Let N be some non-master node in a cascade under normal operating conditions. Then $(j-1)$-synchronization of N implies j-receptiveness of N.*

A detailed proof is given in the appendix. Intuitively, it suffices to show that, given j-receptiveness of $Par(N)$ and $j-1$-synchronization of N, messages will always arrive after the quiet interval has elapsed at N, and before timer of N reaches T_{ma}. It is sufficient to examine two corner cases, where (1) N_0's clock is "fast", N's clock is "slow", the synchronizing message at $j-1$ has maximum latency, and the synchronizing message at j has minimum latency, and (2) N_0's clock is "slow", N's clock is "fast", the synchronizing message at $j-1$ has minimum latency, and the synchronizing message at j has maximum latency.

LEMMA 4. *Under normal operating conditions, all nodes in $\mathcal{N}$ are j-receptive and j-synchronized for all j.*

PROOF. Double induction over the index set for j, and over the nodes on the path from N_0 to given node $N \in \mathcal{N}$, using Lemmas 2 and 3. □

THEOREM 1. *Under normal operating conditions, a non-master node N will always remain in state* EXTERNALLY_TRIGGERED.

PROOF. We observe that N is initialized to state EXTERNALLY_TRIGGERED. Because of Lemma 4, N is j-receptive for all j, we conclude from Definition 11 that the precondition for leaving state EXTERNALLY_TRIGGERED, $\mathsf{timer}_i = T_{ma}$, will never hold. Therefore, the only reachable state for N is EXTERNALLY_TRIGGERED. □

In case of temporary message losses, the network is operating under *transient fault conditions*: nodes affected by loss of their synchronizing message may transition temporarily to state MESSAGE_ABSENT. We can show, however, that a given node will always re-synchronize itself with the master, and count never reaches n_{pf}. As a consequence, the node will never enter state SELF_TRIGGERED.

DEFINITION 13 (TRANSIENT FAULT CONDITIONS). *A cascade is said to operate under* transient fault conditions *iff it is n_{pf}-bounded lossy, and the ε-bounded clock drift assumption holds.*

LEMMA 5. *Let N be some non-master node in a cascade under transient fault conditions. Then if there exists an n, $1 \le n \le n_{pf}$, such that N is $(j-n)$-synchronized, then N is j-receptive.*

Again, the details of the proof can be found in the appendix. It demonstrates that, for any n' such that $1 \le n' \le n_{pf}$, if a node N has performed $n'-1$ unsynchronized cycles, synchronizing messages will arrive after the quiet interval has elapsed, and before timer of N reaches T. Similarly to Lemma 3, the two corner cases are: (1) N_0's clock "fast", N's clock "slow", synchronizing message at $j-n$ with maximum latency, synchronizing message at j with minimum latency, and (2) N_0's clock "slow", N's clock "fast", synchronizing message at $j-1$ with minimum latency, synchronizing message at j with maximum latency.

LEMMA 6. *Let N be some non-master node in a cascade under transient fault conditions. Then for a given step $j > n_{pf}$, let $J = \{j-n_{pf}, \ldots, j-1\}$ be a set of successive step indices. If N and Par(N) are J-receptive, then there is at least one $j'' \in J$ such that N is j''-synchronized.*

PROOF. Transient fault conditions imply that the synchronizing link from N_0 parent to N is n_{pf}-bounded lossy. According to our definition for the node's behavior, N_0 will send a synchronization message in every step. Then from Definition 9, it is clear that N receives a synchronizing message for at least one $j'' \in J$, and so Lemma 6 is a direct adaptation of Lemma 1. □

LEMMA 7. *Under transient fault conditions, all nodes in $\mathcal{N}$ are j-receptive for all j.*

PROOF. The proof is again by double induction over js in the index set, and over the nodes on the paths from N_0 to nodes $N \in \mathcal{N}$.

(1 – base case) $\forall j.j$-receptive(N_0):
By Definition 11.

(2 – induction step) $(\forall j.j$-receptive(Par$(N))) \implies (\forall j.j$-receptive$(N))$:
Split into cases (2a) and (2b) for the inner induction.

(2a – base case)
$\forall j \le n_{pf}.j$-receptive(N):
By Definition 10, N is 0-synchronized. So there exists an $n \le n_{pf}$ such that N is $(j-n)$-synchronized and, By Lemma 5, N is j-receptive.

(2b – induction step)
$\forall j > n_{pf}.\ (\{j-n_{pf}, \ldots, j-1\}$-receptive(Par$(N))$
$\quad \wedge \ \{j-n_{pf}, \ldots, j-1\}$-receptive$(N))$
$\quad \implies \ j$-receptive(N) :
Because N and Par(N) are j'-receptive for all $j' \in \{j-n_{pf}, \ldots, j-1\}$, we know by Lemma 6 that there exists an $n \le n_{pf}$ such that N is $(j-n)$-synchronized. But this is just the precondition for Lemma 5, so N is j-receptive. □

THEOREM 2. *Under transient fault conditions, a non-master node will never enter state* SELF_TRIGGERED.

PROOF. By Lemma 7, the non-master node will be always receptive when receiving a synchronizing message. Therefore, each arriving synchronizing message is received. Transient fault conditions guarantee that the medium loses at most $n_{pf}-1$ subsequent messages. Consequently, count never reaches n_{pf}, so the precondition count $\ge n_{pf}$ for transitioning to state SELF_TRIGGERED will never hold. □

We denote as *permanent fault conditions* all other operating conditions, such as non-n_{pf}-bounded lossy cascades, violation of the ε-bounded clock drift assumption, or complete failure of nodes. Behavior of the cascade under such conditions will not be discussed in the scope of this paper.

4. PROPERTIES

This section defines some essential requirements that a synchronization cascade has to satisfy for distribution of synchronous programs in medium-criticality applications, and demonstrates the corresponding formal properties.

4.1 Requirements

P-reactivity

Distributed real-time control applications typically contain *periodic, reactive* parts which continually compute output values from a given input. Because inputs may originate from other nodes, it is highly desirable to provide an architecture that allows reactive programs to safely synchronize their local processing with communication on the medium, eliminating the needs for special "watchdogs" or similar mechanisms. We will define a property called *P-reactivity* which captures the fact that a node performs communication actions with a certain minimal frequency. Local processing can therefore be triggered by the communicatin handler

DEFINITION 14 (P-REACTIVITY). *A node N_i is called P-reactive for some $P \in \mathbb{R}_+$ iff, for all possible executions of N_i and for all instants t, there is at least one activation instant for a send/receive phase in the time interval $[t, t + P)$.*

Unit delay and length preserving channels

Semantically correct deployment of a synchronous specification warrants that the communication channels provided by the communication layer are valid implementations of the corresponding abstract channels in the specification. As will be indicated in Section 4.4, we will use unit delay channels as our model for an abstract channel. Breaking down the original requirements for the cascade from Section 1 to the implemenation of channels, the cascade should (1) provide a valid implementation of unit delay channels under normal operating conditions, and (2) provide some limited service, including synchronization, under transient fault conditions. The synchronization service can be abstracted as a lossy channel with the 1-length-preserving property. Length-preservation then captures the fact that sender and receiver never get "out of sync".

In the following, we will show that the cascade indeed satisfies the stated requirements.

4.2 Properties of Synchronization Cascades

PROPOSITION 1. *Under normal operating conditions or transient fault conditions, all nodes are $(T_{ma}/(1-\varepsilon))$-reactive.*

By definition of timer_task in Fig. 3, N_i is activated at least every T_{ma} time units (measured by its physical clock) in all of the three possible states EXTERNALLY_TRIGGERED, MESSAGE_ABSENT, SELF_TRIGGERED. Because the bounded clock drift assumption holds under normal and transient fault conditions, the worst case of a "slow" physical clock is $d\mathsf{timer}_i/dt = 1-\varepsilon$. In physical time, the greatest interval in between activations is therefore $T_{ma}/(1-\varepsilon)$. □

PROPOSITION 2. *Under normal operating conditions, the input/output behavior of a synchronizing link is a unit delay channel.*

For an intuitive treatment, there are four parts constituting the unit delay channel property: (1) every message sent by the sender must be accepted by the communication medium, (2) once accepted, the message must reach the receiver, (3) the cascade receiver must be receptive, (4) a message computed at step j at the sender is processed at step $j + 1$ at the receiver. (1) and (2) are guaranteed by the 1-bounded message loss assumption. (3) and (4) are direct consequences of Lemma 4.

PROPOSITION 3. *Under normal operating conditions or transient fault conditions, the input/output behavior of a synchronizing link is a 1-length-preserving channel.*

In the case of normal operating conditions, the 1-length-preserving property results from Proposition 2. For transient fault conditions, the fact that the synchronizing link itself is 1-length-preserving follows from Lemma 6: let N, N' be the sender and receiver of the synchronizing message, respectively. For a given step j, there are two possibilities: (1) if the synchronizing message is not lost in step j, then N and N' will both be j-synchronized. They will therefore agree on the step number, and N' will process in step j the result of N's computation at step $j - 1$, so the channel's behavior may be characterized as a unit delay for step j. (2) if the synchronizing message is lost in step j, then N' will detect a $\perp$ symbol each time N emits a synchronizing message. In both cases, the input/output behavior of the link constitutes a 1-length-preserving channel.

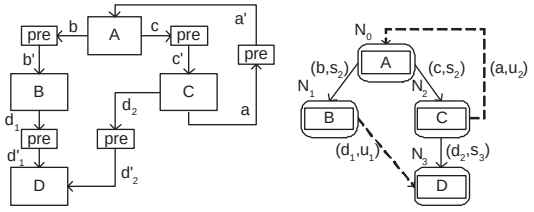
4.3 Nonsynchronizing messages

In the discrete-time abstraction of the synchronous programs, synchronizing messages correspond to messages with *deterministic* timing: if the sender component has computed the synchronizing message at step j, the synchronizing message will always be processed by the receiver component at step $j+1$ in normal operation. This is why abstracting the link as a (deterministic) unit delay channel in the synchronous program, as shown in the Section 4.2, is justified for synchronizing links.

For nonsynchronizing links, the deterministic delay channel abstraction may not always be valid. A nonsynchronizing message computed in step j by a sender node may reach a receiver node at steps j, $j + 1$, ..., depending on the timing of activations of the two nodes, and the timing of messages on the synchronizing link. Fortunately, best/worst-case analysis, such as in [9], can be used in theory to ensure that *some* nonsynchronizing messages have deterministic timing. For other messages, it may be necessary to either add some flow control mechanism to the communication layer, or to account for the nondeterminism in the discrete-time abstraction.

4.4 Mapping Synchronous Programs to Cascades

Consider the simple network in Fig.4(a): The network includes components $\{A, B, C, D\}$, and unit delay channels *pre* in between components. A sends messages through signal b to component B, and through signal c to component C. B sends messages to D (signal d_1), C sends messages through signal d_1 to component D, and to A through signal a. The dataflow network is mapped to the cascade of Fig. 1 with the mapping $\{(A, n_0), (B, n_1), (C, n_2), (D, n_3)\}$ Fig. 4(b) shows a depiction of the resulting cascade. The resulting distribution is correct for normal operation if the two nonsynchronizing channels u_1, u_2 have deterministic unit-delay behavior. Note that the synchronization messages carry values that the distributed program needs to communicate. If, for a given system step, no such value needs to be communicated, an empty message with no relevant data must be used as a synchronizing message.



(a) Dataflow network (b) Deployed dataflow network

Figure 4: Mapping a dataflow network to a cascade

5. CONCLUSION

Our method of distribution relies on the existence of delays at the partitioning boundaries in the specification. Clearly, introduction of such delays is somewhat implementation-driven: an "ideal" platform with infinite resources would not require delays in the model, except for breaking causal loops. Bottom-up introduction of delays will in most cases necessitate a re-validation of the entire design, and is in conflict with the idea of having implementation-independent synchronous specifications. We propose in [1] a methodology which enforces introduction of delays at the boundaries of (abstract) software components at early design stages, thus ensuring both implementation-independence and partitionability of the specification.

In comparison to the LTTA approach of Benveniste et. al. [3], synchronization cascades are designed for protocols with restricted availability, while in LTTA, the bus is assumed to be ideally available. LTTA links may be abstracted as deterministic channels, while in cascades, even under normal conditions, some unsynchronized links may exhibit nondeterministic behavior. In cascades, the activation timing of non-master nodes always depends on the master node, and is roughly periodic. This is also true if synchronous programs with (delayed) feedback loops are deployed onto a cascade. For LTTA networks, convergence of the activation frequencies for periodic, length-preserving programs with feedback is not obvious from [3] [4].

For the special case of the CAN protocol, Time-Triggered CAN (TTCAN) [6] is a CAN-based synchronization layer which, in its Level 1 stage, does not rely on additional hardware for synchronization, similar to cascades. Arbitration in TTCAN is primarily based on static assignment of message slots. An interesting question is whether existing unsynchronized nodes can be integrated with the synchronized network: In TTCAN, unsynchronized nodes may only have read access to the bus, while in a CAN-based cascade, full read/write interoperation is possible if the messages sent by unsynchronized nodes are included in the message jitter analysis in Section 3.2.

High-precision clock synchronization algorithms such as [10] are well-studied. This kind of algorithm provides a high-precision synchronization, where clocks are synchronized within an interval in the range of $\Delta(1 - 1/|\mathcal{N}|)$, with $|\mathcal{N}|$ the number of nodes, while cascades are merely synchronized in the range of $T/2$, with T as the base period. However, we claim that our design of a synchronization cascade is more specifically suited to the requirements outlined in Section 1: For instance, Welch and Lynch's algorithm requires $|\mathcal{N}|^2$ synchronization messages for each round vs. $|\mathcal{N}| - 1$ messages for a cascade. Welch and Lynch's algorithm uses explicit synchronization rounds, where the synchronization messages could potentially block other real-time traffic on the medium. Cascades, on the other hand, provide synchronization using the regular real-time traffic of the distributed program. We also claim that the $T/2$ precision may be sufficient in cases where synchronization is important for correct, timely implementation of the distributed program's semantics, but a precise absolute global time base is not necessary.

Our next goals will be an experimental evaluation of the method along with some tool support [7], and the definition of a fault-modular variant of the cascade, where subtrees within the cascade can safely retain their relative synchronization in the case of master faults.

[4] Consider a periodic synchronous program with (delayed) feedback loop deployed onto an LTTA with nodes N_1, N_2: If the sequence sent by N_1 has periodic timing, the timing of the received, decoded sequence at N_2 is usually aperiodic because N_2's alternating bit decoder will occasionally drop duplicate messages. For providing a periodic feedback sequence to N_1, N_2 must either send duplicate messages, or adjust its send rate to the average frequency of the decoded sequence, and vice versa for N_1. It is unclear how the actual periods of N_1 and N_2 converge in such cases.

Acknowledgements

Thanks to Bernhard Schätz and the anonymous reviewers for helpful comments on an earlier version of this paper.

6. REFERENCES

[1] A. Bauer and J. Romberg. Model-based deployment: From a high-level view to low-level code. In *Proceedings of the 1st International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, Hamilton, Canada, June 2004.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 2003.

[3] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proceedings of EMSOFT 2002*. Springer-Verlag, 2002.

[4] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer, 2001.

[5] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, May/June 1999.

[6] T. Führer, B. Müller, F. Hartwich, and R. Hugel. Time triggered CAN (TTCAN). In *SAE 2001*, Detroit. SAE number 2001-01-0073.

[7] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[8] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, Boston, 1997.

[9] K. Tindell and A. Burns. Guaranteeing message latencies on controller area network (CAN). In *Proceedings 1st International CAN Conference*, September 1994.

[10] J. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

APPENDIX

Upper and Lower Bounds

We will use the following properties for working with upper and lower bounds: Let S be a set, let $F, G : S \to \mathbb{R}$ be functions from S to the reals such that upper and lower bounds exist for F, G, and let $\min_{s\in S}(F(s))$ and $\max_{s\in S}(F(s))$ be the lower and upper bounds of F on S, respectively. Furthermore, let C be some constant. Then the following properties hold:

$$\max_{s\in S}(C) = C \quad (8)$$

$$\min_{s\in S}(C) = C \quad (9)$$

$$\max_{s\in S}(F(s)+G(s)) \leq \max_{s\in S}(F(s)) + \max_{s\in S}(G(s)) \quad (10)$$

$$\min_{s\in S}(F(s)+G(s)) \geq \min_{s\in S}(F(s)) + \min_{s\in S}(G(s)) \quad (11)$$

$$\max_{s\in S}(-F(s)) = -\min_{s\in S}(F(s)) \quad (12)$$

$$\min_{s\in S}(-F(s)) = -\max_{s\in S}(F(s)) \quad (13)$$

$$\max_{s\in S}(F(s)) < \min_{s\in S}(G(s)) \implies \forall s \in S.F(s) < G(s) \quad (14)$$

$$\max_{s\in S}(F(s)) \leq \min_{s\in S}(G(s)) \implies \forall s \in S.F(s) \leq G(s) \quad (15)$$

Proof of Lemma 3

For j-receptiveness of N, Equations 5 (case (1)) and 6 (case (2)) must hold.

(1) We can rewrite Equation 5 as

$$Q + (t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + (t_{0,j} - t_{0,j-1}).$$

This condition is quantified over all possible executions of the cascade under normal operating conditions: we indicate the set of such executions with NOC. Using Equations 8–15, we eliminate the quantification and use lower/upper bounds instead:

$$Q + \max_{(NOC)}(t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + \min_{(NOC)}(t_{0,j} - t_{0,j-1}).$$

From Section 3.2, it follows that $T/(1+\varepsilon)$ is a lower bound for $t_{0,j} - t_{0,j-1}$. Because N is $(j-1)$-synchronized by assumption, the bounded message jitter property from Section 3.2 yields $\max_{(NOC)}(t_{i,j-1} - t_{0,j-1}) = d_{\max}(0,i)$. Substituting and using Equation 1, the following must hold:

$$Q + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < \dfrac{T}{1+\varepsilon}.$$

Using the $(T-Q)/2$ bound for the message jitter from Definition 6, we have to show

$$Q + \frac{T-Q}{2} < \frac{T}{1+\varepsilon}.$$

Solving for Q yields the condition ($0 \leq \varepsilon < 1$)

$$Q < T\left(\frac{1-\varepsilon}{1+\varepsilon}\right),$$

which holds for $Q < T \cdot (1-2\varepsilon), 0 \leq \varepsilon < 1$. This proves case 1.

(2) By assumption, it is true that $(j-1)$-synchronized(N). Rewriting Equation 6 and using upper/lower bounds yields:

$$\frac{T_{ma}}{1+\varepsilon} + \min_{(NOC)}(t_{i,j-1} - t_{0,j-1}) > \max_{(NOC)}(t_{0,j} - t_{0,j-1}) + d_{\max}(0,i)$$

With $T/(1-\varepsilon)$ as an upper bound for $t_{0,j} - t_{0,j-1}$, $d_{\min}(0,i)$ as a lower bound for $(t_{i,j-1} - t_{0,j-1})$, and Equation 1, we obtain:

$$\frac{T_{ma}}{1+\varepsilon} > \frac{T}{1-\varepsilon} + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j).$$

Substituting our choice for T_{ma} from equation 2 and solving for $\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)$ with $0 \leq \varepsilon < 1$ yields:

$$\textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < \dfrac{T(1-5\varepsilon)}{2(1-\varepsilon^2)}.$$

For $0 \leq \varepsilon < 1$, this inequation follows from Definition 6. This proves case 2. □

Proof of Lemma 5

By assumption, there exists an $n \in \{1, \ldots, n_{pf}\}$ such that N is $(j-n)$-synchronized. Let n' be the smallest such n. Then for j-receptiveness of N, Equations 5 (case (1)) and 6 (case (2)) must hold.

(1) We distinguish cases (1a) (N is $(j-1)$-synchronized) and (1b) (N is $(j-n')$-synchronized, and $2 \leq n' \leq n_{pf}$).

(1a) See case (1) of the proof for Lemma 3.

(1b) We can rewrite Equation 5 as

$$Q + (t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + (t_{0,j} - t_{0,j-1}).$$

The equation is implicitly quantified over the set of executions under transient fault conditions, TFC. Quantification is removed by taking upper/lower bounds:

$$Q + \max_{(TFC)}(t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + \min_{(TFC)}(t_{0,j} - t_{0,j-1}). \quad (16)$$

For obtaining an upper bound for $t_{i,j-1} - t_{0,j-1}$, we split the term $t_{i,j-1} - t_{0,j-1}$ using the identity

$$\begin{aligned} t_{i,j-1} - t_{0,j-1} = \; & (t_{i,j-n'} - t_{0,j-n'}) \\ & + (t_{i,j-1} - t_{i,j-n'}) \\ & - (t_{0,j-1} - t_{0,j-n'}). \end{aligned}$$

Taking the maximum over executions TFC, and using Equations 8–13, we obtain

$$\begin{aligned} \max_{(TFC)}(t_{i,j-1} - t_{0,j-1}) \leq \; & \max_{(TFC)}(t_{i,j-n'} - t_{0,j-n'}) \\ & + \max_{(TFC)}(t_{i,j-1} - t_{i,j-n'}) \\ & - \min_{(TFC)}(t_{0,j-1} - t_{0,j-n'}), \end{aligned}$$

which can be resolved as follows:

- $\max_{(TFC)}(t_{i,j-n'} - t_{0,j-n'})$:
  We observe that, by assumption of Lemma 5 and using the right-hand side condition of Equation 4, $t_{i,j-n'} - t_{0,j-n'} \leq d_{\max}(0,i)$. Therefore, $d_{\max}(0,i)$ is a valid upper bound.
- $\max_{(TFC)}(t_{i,j-1} - t_{i,j-n'})$:
  According to the operational definition of N, N will first detect a message absence (yielding $T_{ma}/(1-\varepsilon)$ as an upper bound for the duration of cycle $j-n'$) and then perform $n'-2$ unsynchronized steps (yielding an upper bound of $(n'-2)\cdot T/(1-\varepsilon)$ for the remaining cycles). The total upper bound is $T_{ma}/(1-\varepsilon) + (n'-2)\cdot T/(1-\varepsilon)$.
- $\min_{(TFC)}(t_{0,j-1} - t_{0,j-n'})$:
  The lower bound for the duration of $n'-1$ cycles of the master is $(n'-1)\cdot T/(1+\varepsilon)$.

We substitute the upper bound for $t_{i,j-1} - t_{0,j-1}$ into Equation 16, and use $T/(1+\varepsilon)$ as a lower bound for $t_{0,j} - t_{0,j-1}$ and Equation 1 for $d_{\max} - d_{\min}$. Then the following property remains to be shown:

$$\frac{T_{ma}}{1-\varepsilon} + (n-2)\frac{T}{1-\varepsilon} + Q + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < n'\dfrac{T}{1+\varepsilon}.$$

Solving for n' results in the condition ($0 \leq \varepsilon < 1, T > 0$):

$$n' < \frac{1}{4T\varepsilon}\left(T(1+\varepsilon) - \left(2Q + 2\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right)(1-\varepsilon^2)\right).$$

For $0 \leq \varepsilon < 1$, this holds if

$$n' < \frac{1}{4T\varepsilon}\left(T - \left(2Q + 2\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right)\right).$$

This follows from $n' \leq n_{pf}$ and Equation 3, so we're done for case 1.

(2) We distinguish cases (2a) (N is $(j-1)$-synchronized) and (2b) (N is $(j-n')$-synchronized and $2 \leq n' \leq n_{pf}$).

(2a) See case (2) of the proof for Lemma 3.

(2b) For this case, it is true that $\neg(j-1)$-synchronized(N). Rewriting Equation 7 and using upper/lower bounds yields:

$$\frac{T}{1+\varepsilon} + \min_{(TFC)}(t_{i,j-1} - t_{0,j-1}) > \max_{(TFC)}(t_{0,j} - t_{0,j-1}) + d_{\max}(0,i). \tag{17}$$

A lower bound for $t_{i,j-1} - t_{0,j-1}$ is again found by splitting up the term and using Equations 8–13:

- $\min_{(TFC)}(t_{i,j-n'} - t_{0,j-n'})$:
  By assumption of Lemma 5 and using the left-hand side condition of Equation 4, $t_{i,j-n'} - t_{0,j-n'} \geq d_{\min}(0,i)$, so $d_{\min}(0,i)$ is a valid lower bound.
- $\min_{(TFC)}(t_{i,j-1} - t_{i,j-n'})$:
  N will first detect a message absence (lower bound $T_{ma}/(1+\varepsilon)$) and then perform $n'-2$ unsynchronized steps (lower bound $(n'-2)\cdot T/(1+\varepsilon)$ for the remaining cycles). The total lower bound is $T_{ma}/(1+\varepsilon) + (n-2)\cdot T/(1+\varepsilon)$.
- $\max_{(TFC)}(t_{0,j-1} - t_{0,j-n'})$:
  The upper bound for the duration of $n'-1$ cycles of the master is $(n'-1)\cdot T/(1-\varepsilon)$.

With $T/(1-\varepsilon)$ as an upper bound for $t_{0,j} - t_{0,j-1}$ and Equation 1, substituting the above bounds into Equation 17 yields:

$$\frac{T_{ma}}{1+\varepsilon} + (n'-1)\frac{T}{1+\varepsilon} > n'\frac{T}{1-\varepsilon} + \sum_{s_j \in Li(N_i)} \Delta_{li}(s_j).$$

Solving for n' yields ($0 \leq \varepsilon < 1, T > 0$):

$$n' < \frac{1}{4T\varepsilon}\left(T(1-\varepsilon) - 2\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)(1-\varepsilon^2)\right).$$

For $0 \leq \varepsilon < 1$, this constraint is satisfied if

$$n' < \frac{1}{4T\varepsilon}\left(T(1-\varepsilon) - 2\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right).$$

Again, for $n' \leq n_{pf}$, this follows from Equation 3. This concludes the proof for case 2. □