Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture,
22/11/2019. https://doi.org/10.1016/j.sysarc.2019.101690

Asynchronous vs. Synchronous Interfacing to Time-Triggered Communication Systems

Peter Puschner
Vienna University of Technology
Vienna, Austria
peter@vmars.tuwien.ac.at

Raimund Kirner
University of Hertfordshire
Hatfield, United Kingdom
r.kirner@herts.ac.uk

Abstract

Time-triggered communication facilitates the construction of multi-component real-time systems whose components are in control of their temporal behaviour. However, the interface of a time-triggered communication system has to be accessed with care, to avoid that the temporal independence of components gets lost. This paper shows two interfacing strategies, one for asynchronous interface access (in two variants, one being the new Rate-bounded Non-Blocking Communication *protocol) and one for time-aware, synchronized interface access, that allow components to maintain temporal independence. The paper describes and compares these interfacing strategies.*

1 Introduction

In many safety- and time-critical applications (e.g., powerplants, medical devices, planes), an increasing number of functions are realized by embedded computer systems. As a consequence, embedded computer systems get more complex. They need more computational resources and are implemented as distributed systems, consisting of an ever-growing number of computer units. Given this growth, we have to ensure that the additional interactions of components do not create interferences that make the system behavior unpredictable [17], thus infringing the safety of applications.

A composable system design limits the interferences between the components (see discussion on near decomposability in [16]) of a distributed embedded system [7]. The components of a composable multi-component system are highly autonomous: the point of control that determines *which* actions a component performs and *when* these actions are triggered resides *within*, not outside the component. Component autonomy ensures that components and multi-component subsystems can be developed independently. Their functionality and timing can be verified and validated in isolation, thus cleanly separating the responsibilies of the suppliers of different subsystems.

Prior work has shown that time-triggered communication allows components of multi-component systems to autonomously control their timing behavior [4]. Reading from or writing to a time-triggered communication interface is similar to reading or writing program variables that are regularly updated. As opposed to event-triggered interfaces, where every received message has to be read and consumed in order to keep the receiving component in a consistent state (i.e., the receiver must read/process every message within a defined time after its arrival), time-triggered communication does not impose control pressure (or variable load) on components whose interface data get updated [2, 6]. Messages can be classified as either event-messages, state messages, or semi-state messages [10]. Time-triggered communication is suitable for state messages or semi-state messages.

While the program-variable semantics of time-triggered interface data does not create an external control pressure on components from the side of the communication system, mutual-exclusion blocking, which must be enforced when the communication system and a component access the interface concurrently, might still lead to external control on the blocked subsystem.

In this paper we explore two approaches to avoid external influences on control due to mutual-exclusion blocking when accessing the data-sharing interface of a time-triggered communication system. The contributions of this article are as follows:

- Identification of two fundamental access strategies for time-triggered communication interfaces, namely asynchronous and synchronous access.

- Comparing these two access strategies in the light of replica-determinism and fault tolerance.
- Introducing the *Rate-bounded Non-Blocking Communication* (RNBC) protocol for asynchronous time-triggered communication interfaces. RNBC is more predictable than the existing NBW protocol [8], while also having a smaller implementation overhead.
- Providing a schedulability criterion for RNBC and NBW with formal proof.
- Providing an extended version of RNBC, based on buffer arrays, to be able to handle more demanding real-time requirements than RNBC can handle.

In the asynchronous approach, subsystems access the interface *asynchronously*, without coordination. The software for interface access prohibits external control on subsystems by masking access conflicts (Section 4.1). In the synchronous approach, computing components that access the time-triggered communication interface are *time-aware*. They synchronize to the global time of the communication system and take advantage of the statically available information about the send and receive times of messages to avoid control conflicts when accessing the interface (Section 4.2). After introducing each of these two approaches for interface access, Section 5 compares the two strategies in detail. In this comparison, particular attention will be given to how each of the interfacing strategies supports replica determinism, as replica determinism is essential for the construction of dependable, safety-critical real-time applications. Section 6 shows exemplary behaviour of the different access strategies.

2 Time-Triggered Communication

A *time-triggered communication system (TTCS)* is an autonomous subsystem of a distributed real-time computer system that transports state messages between the nodes of the computer system in a time-predictable way [3]. Time-triggered messages have a-priory known sending times and are typically sent periodically. The TTCS transports the messages from a sender node to one or more receiver nodes according to a static message-transmission schedule that is constructed at design time. The clock synchronization service of the TTCS provides a global clock to the distributed system. Communication end points of the TTCS, the *linking-interface subsystems (LIFSS)* of the nodes, use this global clock to maintain a uniform view about the progress of time and coordinate their message send and receive operations according to the message schedule. Further, the *computational components (CC)* of the nodes can program the LIFSS to generate (periodic) clock interrupts. This allows computational components to synchronize their operation to the global clock (see e.g., [14]).

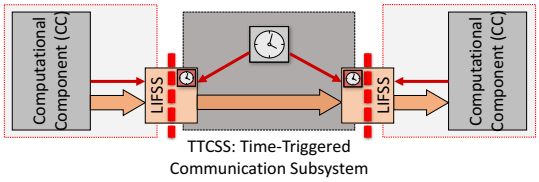

Figure 1. Time-Triggered System Model.

Figure 1 sketches the structure of a time-triggered distributed computer system. The nodes of the distributed system consist of the interacting, autonomous *computational components (CCs)* and the LIFSS. The LIFSS of all nodes and the communication network taken together constitute the TTCS.

2.1 Fault Tolerance

Since systems of high time-predictability are typically needed for building safety-critical systems, we discuss in the following fault tolerance for time-triggered systems.

Time-triggered architectures like the TTA [5] provide support for fault tolerance to address the demands of safety-critical real-time systems. In order to cope with internal physical faults under the single-fault hypothesis, both the transmission medium and the nodes of a time-triggered system can be replicated to form fault-tolerant units. Fault-tolerant units (FTUs) consist of actively replicated nodes that either ensure error detection via voting or error masking by the implementation of self-checking mechanisms that make nodes fail-silent (i.e., the nodes deliver either correct results or no results at all).

As a prerequisite for the realization of the described mechanisms of active redundancy, the behavior of the nodes must be replica-deterministic [12, 13], i.e., given an equal set of timed inputs, replicated nodes visit the same externally visible state and produce the same output messages at points in time that are at most an interval of d time units apart [4].

The TTA provides replica determinism at the LIFSS of the communication subsystem. It is, however, up to the designer and developer of the CC to ensure the replica determinism for the whole node. Within this paper we will show how the way of interfacing to the LIFSS influences replica determinism. This will be of

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

central interest when judging and comparing the two interfacing strategies proposed in this paper with respect to their adequacy for safety-critical applications.

3 Autonomy of Components

In a general distributed computer system, individual components communicate via the exchange of messages. Besides the data flow, we have to establish appropriate *control flow* relations between communicating components when we aim at constructing systems that guarantee component autonomy. The control flow determines who initiates message transfers and is in command of the individual steps of the transfer. Let us consider the transfer of a message from a sending to a receiving component. If the sender is entirely in control of the message transfer, then the control flow originates at the sender and terminates at the receiver. We call this an *information-push* [4] operation. If the receiver is in control of transferring the message from a sender, then the control-flow direction is from the receiver to the sender, in opposite direction of the data flow. We call this an *information-pull* [4] operation.

Information-push operations are ideal for senders. For an information-push operation, the sender does not have to wait until the receiver is ready, neither does it need buffers for storing messages while waiting. Information-pull operations are ideal for receivers. Following the information-pull policy, receivers can process messages under their own control – message-pull receivers cannot be disturbed by messages that arrive at times that are not under their own control.

In a TTCS, there is no explicit control flow across the communication system. Due to the program-variable semantics of state messages, message arrival does not impose external control on CCs. Consequently, CCs may, in principle, read ports of the LIFSS in information-pull operations and write to ports in information-push operations at any time. The only potential control conflicts at a LIFSS can arise due to mutual-exclusion blocking when the LIFSS and a CC try to access a shared data item at the same time. We will show how these mutual-exclusion control conflicts can be resolved by taking advantage of the properties of time-triggered communication respectively by using LIFSS services.

4 Non-blocking Interfaces

In the following we will discuss the two alternative strategies of accessing the LIFSS without control disturbances due to mutual-exclusion blocking. CCs may adopt one of these strategies. The chosen strategy determines how and when CCs access the LIFSS ports and whether they use the clock interrupt service of their LIFSSs. Depending on application requirements, CCs may use the data-sharing interface of the LIFSS either as a

- time-agnostic, asynchronous interface, or as a
- time-aware, time-synchronized interface.

In the following we discuss these two different LIFSS interfacing strategies in detail.

4.1 Time-Agnostic, Asynchronous Interface

Assuming that network communication is time-triggered, but the program activation on the computational component (CC) is event-controlled, a time-agnostic asynchronous communication protocol is required at the LIFSS. In this section we introduce the novel *Rate-bounded Non-Blocking Communication Protocol* (RNBC) as such a protocol.

RNBC allows the parallel write and read of shared data without the need for blocking or waiting, supporting one writer and multiple readers. To facilitate non-blocking, RNBC comes with a schedulability criterion that bounds the rate of write accesses. Reading via RNBC always provides the latest completely written data. Any pending data write becomes only available for reading, once the write is completed.

RNBC has been inspired by the *Non-blocking Write Protocol* (NBW) [8]. To understand the benefit of RNBC, we first briefly describe NBW, as shown in Figure 2. The data to be communicated is stored in the buffer `buff`. In addition, the writer maintains the concurrency control flag `ccf`, to indicate the writing status. Whenever `ccf` is odd, a writing is in progress, and when `ccf` becomes even again (and incremented by 2), the writing has completed. Thus, the reader performs a busy waiting during reading to make sure that `ccf` was even and did not change during the whole read operation.

While NBW is simple to understand, part of its nature is that it has a variable execution time, depending on whether the read operation overlaps with a write operation or not. To reduce the likelihood of such a read delay, the authors included in the original paper also an extended variant of NBW, which manages a ring buffer [8]. This way, a write operation started during a read does not necessarily cause a delayed reading. Only in case multiple writes fill up the ring buffer during a pending read, then a delayed read will happen. It has to be noted that this Extended NBW can still

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

```
int buff; // shared msg buffer
int ccf = 0;  // 0 means empty buffer

void nbw_write_msg(int msg) {
  ccf++;  // becomes odd: write in progress
  buff = msg;
  ccf++;  // becomes even: write completed
}

int nbw_read_msg() {
  int msg;
  int ccf1,ccf2=0;
  do {
    ccf1 = ccf;
    if (odd(ccf1)) continue;
    msg = buff;
    ccf2 = ccf;
  } while (ccf1 != ccf2);
  return msg;
}
```

Figure 2. NBW: Classical Non-Blocking Write Protocol

suffer from occasional read delays, however, making them more rare. But at the same time the Extended NBW has a significantly higher code overhead than the standard NBW, making read/write clashes more likely. These given limitations of NBW and its extended ring-buffer variant motivated us to develop the Rate-bounded Non-Blocking Communication Protocol (RNBC), a new concurrent read-write protocol that is inspired by the field of lock-free communication data structures, for example, lock-free message queues [1, 9].

4.1.1 RNBC: A Rate-bounded Non-Blocking Communication Protocol

The design strategy for RNBC was, instead of aiming to minimise the likelihood of read/write clashes, to develop a precise schedulability criterion that guarantees the absence of read/write clashes. This at the same time allowed us to develop a protocol that has minimal code overhead.

The code of RNBC is shown in Figure 3. The key feature of RNBC is to have two communication buffers, one for the current write operation, and the other one for current read operations. This concept of double buffering is very common and used for different purposes, e.g., to smoothen the output of computer graphics [11, 15]. Whenever a write finishes, the roles of these two buffers are swapped. The shared flag `wbuff` indicates which buffer index (0 or 1) is currently allocated for next writing. As we can see from the code, the implementation of both the reader and writer in RNBC is extremely simple, consisting basically just of the access/update of the buffer and the control flag.

```
int buff[2];       // shared msg buffer
int wbuff = 0;     // buffer index of write

void rnbc_write_msg(int msg) {
  buff[wbuff] = msg;
  wbuff = 1-wbuff; // swap read/write buffer
}

int rnbc_read_msg() {
  int rbuff = 1-wbuff;
  return buff[rbuff];
}
```

Figure 3. RNBC: Rate-bounded Non-Blocking Communication Protocol

4.1.2 Schedulability Criterion of RNBC

The strong contribution behind RNBC is the precise schedulability criterion that guarantees the absence of read/write clashes regardless of the relative phase between read and write operations. Before introducing this schedulability criterion, we have to first introduce some definitions:

c_r: the worst-case execution time (WCET) of the read operation

c_w: the WCET of the write operation

$mint$: the minimum inter-arrival time between two messages, i.e., consecutive write operations

Based on these definitions, we can state the schedulability criterion of RNBC:

Theorem 4.1 *Without any further assumption about the synchrony between read and write, the following is a necessary and sufficient schedulability condition for the RNBC protocol with non-preemptable read and write:*

$$c_w + c_r \leq mint \tag{1}$$

This schedulability criterion implies that the maximum execution time $c_{r,max}$, available for a read operation, is as follows: $c_{r,max} \leq mint - c_w$. In other words, we have to make sure that the worst-case execution time (WCET)[18] of the read and write operation together is less or equal than the minimum inter-arrival time of messages. The write and read operations have

to be non-preemptive, so that c_w and c_r do limit the access time to the shared variables.

Proof (Theorem 4.1) To prove this, we have to show that the scheduleability criterion is both sufficient and necessary. To do so, we use Figure 4 to visualise some key properties of the behaviour of RNBC. The first row *write* shows in which buffer the individual write operations are writing (with distance *mint* between them). The 2nd row *wbuff* shows the timing diagram of the flag `wbuff`, following directly from the implementation. The row *rbuff* shows the behaviour of a read operation, depending on its starting time in comparison to write operations. This means that the flag `rbuff` at the start of a read operation is always set as the opposite of the current value of the `wbuff` flag. This line shows that after the completion of a write operation to buffer b_i, any read operation started afterwards and before the next writing operation starts, will read from b_i. The lines *rwindow* b_0 and *rwindow* b_1 show the possible reading window for buffer index 0 respectively 1, such that no read/write clash will occur. The reading window for each buffer b_i spans from the completion of a write into b_i till the beginning of the next write operation into b_i.

Part 1: Sufficient: As shown in Figure 4, the read access from buffer b_i can only start after completion of its write till the next write completion of buffer b_{1-i}, which is a time span of *mint*. At the same time, the reading window of buffer b_i starts as well directly after the completion of its write and lasts till the beginning of the next write operation to buffer b_i, which is a time span of $2mint - c_w$. Any valid read to b_i has to start within the state $rbuff == b_i$ and has to complete before the end of the reading window *rwindow* b_i, for which the minimum duration is:

$$length(rwindow) - length(rbuff)$$

which is

$$(2mint - c_w) - mint$$

and can be simplified to

$$mint - c_w$$

Hence, the minimum guaranteed time available for reading is $mint - c_w$, which implies that $c_r \leq mint - c_w$ is sufficient for non-clashing read/write operations. This proves that the schedulability condition of Theorem 4.1 is sufficient.

Part 2: Necessary: To proof that the schedulability condition is necessary, we use an indirect proof. We start with the assumption that the schedulability condition does not hold and derive from it that the read/write operations can clash.

Assuming that the schedulability condition does not hold, we have the following property:

$$c_w + c_r = mint + \Delta \mid \Delta > 0$$

To look for the worst case, we assume that the read access to buffer b_i starts just at the end of state $rbuff == b_i$, i.e., when *rbuff* has been directed to b_i for the duration *mint*, which is the last possible time before reading is switched to buffer b_{1-i}. In that situation the remaining time of reading window *rwindow* b_i is $mint - c_w$, because $length(rwindow) - length(rbuff) = (2mint - c_w) - mint = mint - c_w$).

However, based on the assumed property $c_w + c_r = mint + \Delta \mid \Delta > 0$ it follows that the reading time c_r is $c_r = mint - c_w + \Delta$. Thus, the reading time of buffer b_i is by Δ longer than the remaining length of the reading window *rwindow* b_i, causing a clash of the read/write operations to buffer b_i. This proves that the schedulability condition of Theorem 4.1 is necessary. □

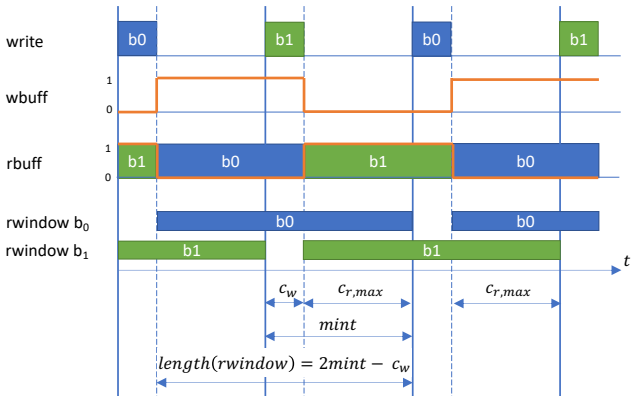

Figure 4. Properties of RNBC Protocol (used in proof of Theorem 4.1)

Corollary 4.2 *If the condition* $(c_w + c_r \leq mint)$ *can be ensured, then there is no benefit in terms of reducing read/write clashes by using more than two communication buffers, e.g., a ring buffer with* $n > 2$ *elements.*

4.1.3 NBW vs RNBC

We have stated that RNBC relies on some real-time assumptions for its correctness: the defined inter-arrival time *mint* and WCETs c_w, c_r. NBW does not have such requirements specified, thus NBW seems to be

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

easier to deploy. However, we herewith compare the protocols in further detail.

As stated in Theorem 4.1, the border-line case where RNBC still works is $c_w + c_r = mint$.

Now lets assume that the WCETs of the read and write operations of NBW are denoted as c'_r and c'_w. Comparing the implementations of NBW in Figure 2 and of RNBC in Figure 3, it is clear that NBW has more overhead for both, read and write, i.e., $c'_w > c_w \wedge c'_r > c_r$. Since NBW has only one communication buffer, read and write have to be strictly sequential. From that it that follows that $c'_w + c'_r > mint$. This means, while RNBC still just works for this *mint* value, the same *mint* value will cause NBW to be caught in never-ending read-write clashes, regardless of the phase between read and write. Thus, also NBW has to rely on some real-time assumptions, which are actually even more strict than those required for RNBC.

To derive also a concrete schedulability condition for NBW, we have to split the execution time c'_r of NBW's reader into the following components:

$c'_{r,prim}$ is the execution time of a read without retries

c'_{odd} is the additional execution time for each handling of an odd `ccf` (read attempt overlapped with write)

c'_{inc} is the additional execution time for each handling of the `ccf` incremented by two (complete write happened within read duration)

We use this distinction between c'_{odd} and c'_{inc} to allow us to be more precise on the overall execution time, as the code executed for these two cases slightly varies. Based on that we can formulate the following sufficient and necessary schedulability condition for NBW:

$$mint \geq c'_{r,prim} + \max\left(c'_{r,odd} \cdot \left\lceil \frac{c'_w + c'_{r,odd}}{c'_{r,odd}} \right\rceil, c'_{r,inc}\right) \tag{2}$$

This criterion has $c'_{r,prim}$ as the read's basic execution time. The schedulability criterion makes sure that the total execution time of NBW's read including potentially multiple additional execution times of c'_{odd} or c'_{inc} always fits within the interval *mint*. In case of interference with a concurrent write activity, there are two cases possible, expressed by the max statement:

1. For the case that the read attempt partially overlaps with a write (identified by an odd `ccf` value), the reader has to run multiple read attempts till the write is finished, each adding $c'_{r,odd}$ to the reader's execution time.
2. Only in the case that $c'_w < c'_r$ holds, it is possible that the reader encounters a complete write that happened within the read attempt (`ccf` still even but incremented by two). In this case $c'_{r,inc}$ is added to the reader's execution time.

It is worthwhile to note that the case 2 ($c'_{r,inc}$) cannot happen multiple times or in combination with case 1 ($c'_{r,odd}$) for the same read attempt, as this is prevented by the minimum value of *mint*. We do not go into detail why the schedulability condition of NBW is more strict than the one of RNBC given in Equation 1 of Theorem 4.1, as we have already shown in the beginning of this section that NBW demands a larger *mint* value compared to RNBC.

Concluding, RNBC is an efficient implementation for the concurrent single-writer multiple-reader communication pattern, using real-time properties like WCET and minimum inter-arrival time to assure correct behaviour. Under the given schedulability condition, RNBC allows a constant access time for both reader and writer. In contrast, with NBW only the write has a constant access time, but not the read.

4.1.4 Extended RNBC using Multibuffer

As stated by Corollary 4.2, the basic RNBC is free of any read/write clashes as long as one can ensure that the property ($c_w + c_r \leq mint$) holds. In most practical situations that might be the case. However, there might be situations where c_w or c_r are relatively long compared to *mint*, which could be due to long message transfer times, or due to narrow timing constraints, requiring the initiation of new data updates while the previous transfer is still pending. In this section we introduce an extended version of RNBC, called *Extended RNBC* (ExtRNBC) using more than two buffers, so that the timing requirement ($c_w + c_r \leq mint$) can be replaced by a less strict one.

The principle of ExtRNBC is very similar to RNBC, just that a buffer array of arbitrary size is used, resulting in a ring-buffer style access by the read and write operations. The source code of ExtRNBC is given in Figure 5. In the code, the buffer size is specified by the constant `B`, using 3 just for demonstration purpose. Whenever the write finishes the message transfer, it will set the shared write buffer index to the next element along the ring buffer, using the C operator "`?:`" to distinguish between simple index increment and wrap around. Whenever a read starts, it takes the messages from the latest transferred message, regardless of how many messages have been written since the last read.

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

```
#define B 3      // size of ring-buffer
int buff[B];     // shared msg buffer
int wbuff = 0;   // buffer index of write

void rnbc_ext_write_msg(int msg) {
  buff[wbuff] = msg;
  // set write buffer to next buffer:
  wbuff = (wbuff==(B-1)) ? 0 : (wbuff+1);
}

int rnbc_ext_read_msg() {
  // set read buffer to one before current
  // write buffer:
  int rbuff = (wbuff==0) ? (B-1) : (wbuff-1);
  return buff[rbuff];
}
```

Figure 5. Extended RNBC: Multi-buffer

Based on this multi-buffer access, we can state the schedulability criterion of ExtRNBC:

Theorem 4.3 *With a given buffer size* `B` *and no further assumptions about the synchrony between read and write, the following is a necessary and sufficient schedulability condition for the ExtRNBC protocol with non-preemptable read and write:*

$$c_w + c_r \leq (B-1) \cdot mint \tag{3}$$

This schedulability criterion implies that the maximum execution time available for a read operation $c_{r,max}$ is as follows: $c_{r,max} \leq (B-1) \cdot mint - c_w$. In other words, we have to make sure that the WCET of the read and write operation together is less or equal than the minimum inter-arrival time of messages scaled up by $(B-1)$, where B is the number of available communication buffers.

The following proof for Theorem 4.3 uses the same structure as the proof of Theorem 4.1. The only difference is in the time domain, as ExtRNBC has a more relaxed timing requirement than RNBC due to the longer time till a buffer b_i gets overwritten again.

Proof (Theorem 4.3) To prove this, we have to show, both, that the scheduleability criterion is sufficient and also necessary. To do so, we use Figure 6 to visualise key properties of the behaviour of ExtRNBC. While this figure examplifies the behaviour for 3 buffers, the given reasoning is based on the generic case $B \geq 2$. The first row *write* shows in which buffer the individual write operations are writing (with distance *mint* between them). The 2nd row *wbuff* shows the timing diagram of the flag `wbuff`, which is a ring-buffer style increment of the index, following directly from the implementation. The row *rbuff* shows the behaviour of a read operation, depending on its starting time in comparison to write operations. This means that the flag `rbuff` at the start of a read operation is always set to the latest completely written message, computed as `wbuff`-1 wrap-around at zero to $B-1$. This line shows that after the completion of a write operation to buffer b_i, any read operation started afterwards and before the next writing operation starts, will read from b_i. The lines *rwindow* b_0, *rwindow* b_1, *rwindow* b_2 show the possible reading window for buffer indices $0-2$, such that no read/write clash will occur. The reading window for each buffer b_i is from the completion of a write into b_i till the beginning of the next write operation into b_i.

Part 1: Sufficient: As shown in Figure 6, the read access from buffer b_i can only start after completion of its write till the write completion at the next buffer index along the ring-buffer ($b_{(i+1) \ mod \ B}$), which is a time span of *mint*.

At the same time, the reading window of buffer b_i starts as well directly after the completion of its write and lasts till the beginning of the next write operation to buffer b_i, which is a time span of $B \cdot mint - c_w$. Any valid read has to start within the state *rbuff* $== b_i$ and has to complete before the end of the reading window *rwindow* b_i, for which the minimum duration is:

$$length(rwindow) - length(rbuff),$$

which is

$$(B \cdot mint - c_w) - mint$$

and can be further simplified to

$$(B-1) \cdot mint - c_w$$

Hence, the minimum guaranteed time available for reading is $(B-1) \cdot mint - c_w$, which implies that $c_r \leq (B-1) \cdot mint - c_w$ is sufficient for non-clashing read/write operations. This proves that the schedulability condition of Theorem 4.1 is sufficient.

Part 2: Necessary: To proof that the schedulability condition is necessary, we use an indirect proof. We start with the assumption that the schedulability condition does not hold and derive from it that the read/write operations can clash.

Assuming that the schedulability condition does not hold, we have the following property:

$$c_w + c_r = (B-1) \cdot mint + \Delta \ \mid \ \Delta > 0$$

To look for the worst case, we assume that the read access to buffer b_i starts just at the end of state

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

rbuff == b_i, i.e., when *rbuff* has been b_i for the duration *mint*, which is the last possible time before reading is switched to the next buffer ($b_{(i+1) \ mod \ B}$). In that situation the remaining time of reading window *rwindow* b_i is

$$(B-1) \cdot mint - c_w$$

because

$$length(rwindow) - length(rbuff)$$

$$= (B \cdot mint - c_w) - mint$$

$$= (B-1) \cdot mint - c_w$$

However, based on the assumed property $c_w + c_r = (B-1) \cdot mint + \Delta \mid \Delta > 0$ it follows that the reading time c_r is $c_r = (B-1) \cdot mint - c_w + \Delta$. Thus, the reading time of buffer b_i is by Δ longer than the remaining length of the reading window, causing a clash of the read/write operations to buffer b_i. This proves that the schedulability condition of Theorem 4.3 is necessary. □

Corollary 4.4 *ExtRNBC is schedulable for any values of the parameters c_r, c_w, mint by selecting a suitable ring-buffer size B:*

$$\forall c_w, c_r, mint.\ \exists B.\ c_w + c_r \leq (B-1) \cdot mint \quad (4)$$

Corollary 4.4 states that ExtRNBC is schedulable for any c_r, c_w, *mint* values by choosing a sufficiently large ring-buffer size B. However, to keep the memory footprint low, we have to find the minimal possible buffer size B that satisfies the schedulability condition of ExtRNBC:

$$\min B \ \mid \ B \geq \left\lceil \frac{c_w + c_r}{mint} \right\rceil + 1$$

The resulting buffer size is visualised in Figure 7 using the timing requirements expressed as the parameters $x = \frac{mint}{c_r}$ and $y = \frac{mint}{c_w}$. Note that scales of the x/y axis are not linear, but rather labelled by concrete data points. The flat bottom plain represents the case where the buffer size B is equal to 2, i.e., timing requirements for which the normal RNBC protocol would be sufficient. With the configuration $x = y = 0.1$ we get a minimum buffer size $B = 21$. For smaller values of x,y the minimum required buffer size would be even higher.

Should it happen that for the concrete values of c_r, c_w, *mint* a buffer size of $B = 2$ is sufficient, then it is preferable to use the implementation of RNBC instead of ExtRNBC, since RNBC has a slightly lower resource overhead in terms of execution time.

4.1.5 Predictability of Asynchronous Interfaces

An asynchronous interface like RNBC or ExtRNBC offers predictability in the time domain, as the execution time of neither the writer nor the reader has a variability in the execution time. NBW offers the sime time-predictability for the writer, but not for the reader. However, RNBC, ExtRNBC, and NBW all suffer from low predictability in the value domain, as it is non-deterministic which message instance will be obtained at a particular read access. As a consequence, the asynchronous interfaces are less suitable to build replica-deterministic systems, because the non-determinism in the value domain makes it highly challenging to realise replicas that have an identical output in the value domain.

4.2 Time-Aware Interface

The time-synchronized access strategy solves the mutual-exclusion problem at the LIFSS by avoiding access conflicts from the very beginning. To this end, it carries out the read and write operations of CCs to the LIFSS at times that do not coincide with the accesses by the TTCS — the times when CCs must not access the LIFSS are derived from the TTCS message-transmission schedule that is constructed at system-construction time.

4.2.1 Accessing the LIFSS

When designing the software for a CC with time-synchronized LIFSS access, one will pay particular attention to synchronize the time of read and write operations from the very beginning. On top of conflict avoidance, one can even benefit from the timing information contained in the message schedule and activate the task on a CC tailored to the timing of LIFSS-read and write operations of the TTCS: Such a task schedule will ensure that a task that reads data from the LIFSS will be activated shortly after the message with these data has been received and made available by the LIFSS. In a similar way, the scheduler will activate a task that writes to the LIFSS right before the TTCS will transmit the written data.

To align LIFSS-read and write operations with the actual message transfers, the CC needs (a) a real-time task scheduler and (b) a mechanism for adjusting its local clock to the clock of the TTCS. The alignment can be accomplished by means of a static, table-driven scheduler, where the scheduling table guarantees that the CC accesses the LIFSS in time windows that are guaranteed to be conflict-free. The programmable

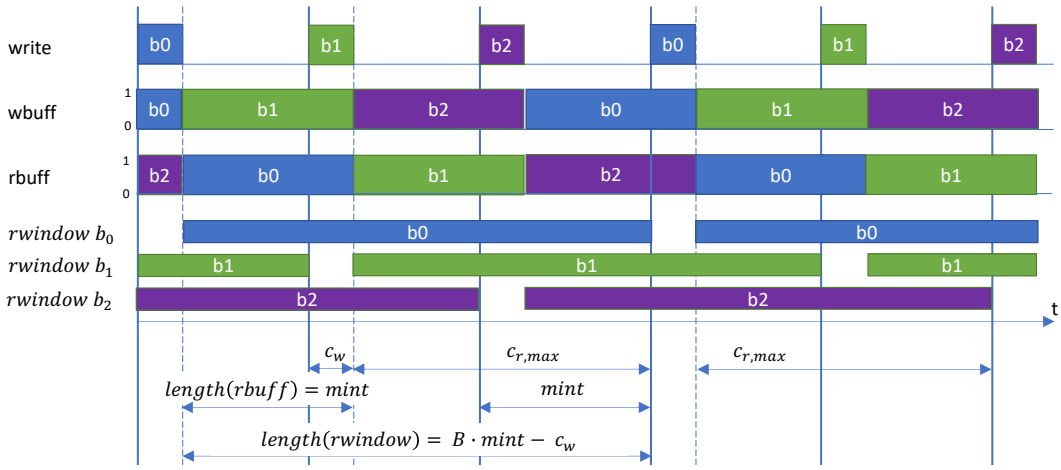

Figure 6. Properties of the Extended RNBC Protocol (ExtRNBC), shown with ring-buffer size $B = 3$ (used in proof of Theorem 4.3)

clock interrupt of the LIFSS can be triggered regularly to synchronize the clock of the CC with the global time-base, i.e., the local representation of the global clock in the LIFSS of a node acts as a master clock for adjusting the clock of the CC of that node in the clock-interrupt handler that is regularly triggered. This clock synchronization ensures that the task scheduler of the CC and the message schedule of the TTCS stay consistent.

4.2.2 Interface-Timing Constraints

Let us provide a more detailed discussion on when a CC may safely access its LIFSS without running into mutual-exclusion conflicts. We assume that the schedule of TTCS accesses to the LIFSS is given. Further we assume that the CC regularly adjusts its clock to the global clock using master-clock synchronization, i.e., in the clock-interrupt handler the clock of the CC is set to the value of the global clock.

In the time intervals between clock synchronization, the clock of the CC may drift away from the global clock. This relative drift has to be taken into account when planning the conflict-free schedule of accesses to the LIFSS. In the following, we will explore how the possible clock drift can be considered when planning the LIFSS access times for CCs. We make the following assumptions:

- The clock of the communication controller that represents the global time of the TTCS acts as the reference clock.
- Tick i of the reference clock happens at time t_i; t_i^j denotes the time of tick i on clock j.
- The drift rate $\hat{\rho}$ denotes the maximum drift rate of a CC clock relative to the reference clock, i.e., $\hat{\rho}$ combines the absolute drift rate of the local representation of the global clock and the maximum absolute drift rate of the clocks of the CCs.

When planning the LIFSS-read and write operations of the CC, we have to ensure that the time intervals of these read and write operations do not overlap with LIFSS-data accesses by the TTCS. The determination of these LIFSS-access intervals has to take the relative drift of the clocks of the CC and the TTCS into account (see Fig. 8).

Let us assume the TTCS accesses some LIFSS data between ticks s and e on its clock, in the time interval $T = [t_s, t_e]$ after the last synchronization point, at time t_0. Now we want to determine the start and end ticks (w and r) of the access interval of the CCs that guarantees mutual exclusion while accounting for the drift of the clocks.

To ensure that all LIFSS accesses by CCs preceding the LIFSS access of the TTCS have completed before T, we must make sure that even the CC with the slowest clock, say CC j, has completed its access at tick w with $t_w^j \leq t_s$, thus taking into account the worst-case drift $\hat{\rho}$ we get

$$w = \left\lfloor \frac{s}{1+\hat{\rho}} \right\rfloor . \tag{5}$$

Analogously, we must guarantee that CCs accessing the LIFSS start only after the TTCS has completed its access. In this case, the CC with the fastest clock, CC k, must not start reading respectively writing the

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

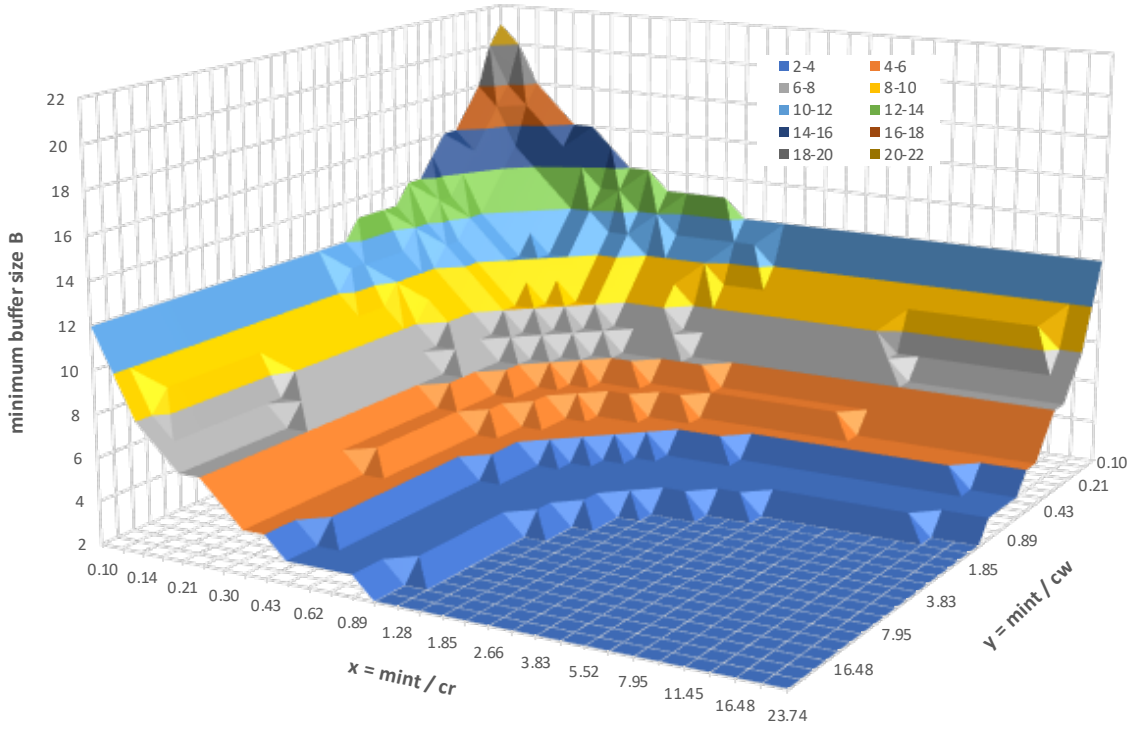

Figure 7. ExtRNBC: Minimum required buffer size B (the bottom plain represents $B = 2$, for which RNBC is suffcient)

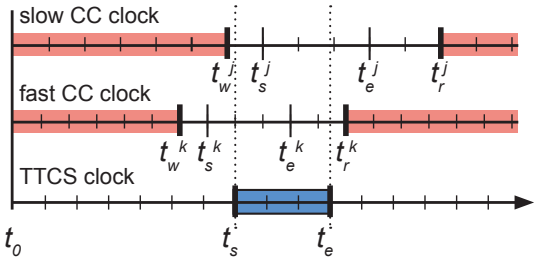

Figure 8. Time window for latest write to and earliest read from LIFSS data.

LIFSS before tick r with $t_r^k \geq t_e$ and

$$r = \left\lceil \frac{e}{1 - \hat{\rho}} \right\rceil . \tag{6}$$

The deviation of the CC clocks from the TTCS clock is smallest after resynchronization and increases over time until the CC clocks are re-synchronized to the TTCS clock again, i.e., the maximum deviation between CC clocks and TTCS clock can be observed right before the end of the resynchronisation interval. So if the duration of the resynchronization interval as timed by the TTCS clock is R_{sync}, then this maximum deviation is $R_{sync} \cdot \hat{\rho}$.

4.2.3 Scheduling CC Tasks

To maintain mutual exclusion, real-time tasks on the CCs that access LIFSS data must be scheduled such that they do not overlap with the respective $[w, r]$ intervals. These mutual exclusion constraints are added to the application-specific precedence constraints of the scheduler. If tasks are statically scheduled, the pre-runtime scheduler uses those constraints together to build the dispatch tables that the dispatchers running on each of the CCs will interpret at runtime.

4.2.4 Interface Timing Properties

Synchronizing the LIFSS accesses of the CCs and the TTCS has the following four positive effects on the timing properties of the interface:

First, synchronizing both LIFSS-write and read operations on all CCs to global time reduces the message-

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

delay jitter in comparison to non-synchronized access. In fact, if we use a synchronized, table-driven scheduler to trigger the LIFSS reads and writes, the jitter can be kept as small as Π, the precision of the global clock.

Second, using table-driven schedules that are aligned to the global clock allows system designers to streamline task executions and message communication. This way, the overall response times of real-time transactions spreading over two or more CCs can be kept short.

Third, if all data manipulations and data transfers via the TTCS follow a global time schedule, then the information about the age of data items is implicitly available at all CCs of the computer systems. As a consequence, the time stamps of real-time observations do not have to be transported via the TTCS. This means, in a time-triggered network with fully time-synchronized CCs, only the *values* of observations have to be handed over to the LIFSS. Both, the *name* and the *time* of the observation are implicitly known.

The final important point is that a synchronized and planned LIFSS access helps us to construct replica-deterministic components, which is crucial for the construction of fault-tolerant systems. As time-synchonized access involves the planning of every access to the LIFSS and enforces a pre-determined order and timing of all read and write operations to the LIFSS, LIFSS accesses by replicated components can be scheduled such that they are guaranteed to work on the same message instances and inputs within the needed time intervals.

5 Comparison

Each of the presented access strategies to the interface of a time-triggered communication system aims at the automomy of CCs, to make components time-composable. The different strategies are, however, paired with significant differences in the characteristics of the components that constituate the overall system. We discuss the differences of the component characteristics in the following; see also Table 1.

A CC that accesses the LIFSS as a *time-agnostic interface* acts fully autonomously, i.e., it ignores the message schedule and potential conflicts when interacting with the LIFSS. A *time-aware component*, in contrast, uses its knowledge about the message schedule to synchronize its LIFSS accesses to the operation of the TTCS.

5.1 Message-Validity Jitter

The fact that components with a time-agnostic interface do not synchronize with the message schedule leads to a message-validity jitter of (almost) two times the message period for all data items read from the LIFSS. This is due to the fact that at both, the sender side and the receiver side, the duration of the time interval between the LIFSS access of the TTCS and access of the CC may vary between zero and the duration of a message period. This jitter is significantly higher than for time-aware interfaces, for which it is Π, the precision of the global clock.

5.2 Execution-Time Jitter

Write operations to the LIFSS always have a constant execution time. Regarding read operations in NBW, there is an execution-time jitter due to the possibility of retries. Both, RNBC and the time-synchronized reads have no execution-time jitter due to LIFSS-access conflicts. This means that the use of RNBC allows for a construction of components that are fully autonomous, without any external control influences on their temporal behavior.

5.3 End-to-End Delay

The lack of synchronization at time-agnostic interfaces inhibits the streamlining (i.e., tight synchronization) of tasks that read from or write to the LIFSS and the messages that transport the respective data items. A time-aware interface with time-synchronized interface access, in contrast, allows for the synchronization of these activities. Thus, time-aware interfaces support real-time transactions with much shorter end-to-end delays than asynchronous interfaces.

When all operations of the components and the communication system are time-triggered, controlled by a global execution plan, then the knowledge about the points in time of all data-read and write operations are globally known, i.e., the timestamps of these operations are implicit global knowledge in the system. As a consequence, and in contrast to systems which operate asynchronously, one does not need to transport the times of real-time observations in time-synchronized systems.

5.4 Replica Determinism & Fault Tolerance

Synchronizing the LIFSS-read and write operations of a component with global time is a prerequisite for a clear definition of the state of a CC, which, in turn, is

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

Table 1. Comparison of the Component Characteristics for the Interfacing Methods.

Characteristic	*Interface*		
	Time Agnostic		*Time Synchronized*
	NBW	*RNBC/ExtRNBC*	
Control paradigm	full autonomy	full autonomy	adaptation to schedule
Message validity jitter	2× message period	2× message period	precision of clock
Jitter of message read	0−2 failed read attempts	0	0
Task-msg. streamlining	no	no	yes
Use of time	value (explicit)	value (explicit)	control (implicit)
Replica determinism	no	no	yes
Build complexity	low	low	medium to high
Example applications	movie streaming, sensor network		flight control, steel mill, drive by wire, film stretching

needed for the realization of replica-deterministic components (i.e., components that agree both in the value and time domain within a defined time span). As the provision of replica determinism greatly simplifies the construction of fault-tolerant real-time systems, time-aware LIFSS access of CCs is as well essential for implementing fault tolerance.

As time-agnostic interface access is asynchronous to the operation of the TTCS, redundant CCs may read or write different data instances when accessing the interface, thus obstructing replica determinism. E.g., due to race conditions caused by slightly different quartz oscillation rates at different nodes, it may happen that one CC reads interface data before they are updated by the LIFSS, while the read operation of a redundant CC returns the data after they have been updated by a new message. In constrast, when interface accesses by CCs are aligned to the operation of the LIFSS, one can ensure that all redundant CC copies behave consistently when sending or receiving data, i.e., they always send and receive the same data and extend the replica determinism provided by the TTCS to the component and application level. This greatly simplifies the construction of real-time systems that are fault tolerant.

5.5 Build Complexity

The central advantage of using interfaces without care about timing is the low build complexity of the components and the overall system. When using time-agnostic interfaces, developers do not need to know about the temporal particularities of the LIFSS or other components. In contrast, to fully exploit the benefits of synchronized interface access, the timing of operations on the components and message transmission on the TTCS have to be tightly coordinated. The complexity and cost for designing such a system-wide coordination may be significant.

5.6 Example Applications

Typical example applications for non-synchonized interfaces are entertainment systems, like movie streaming, or monitoring systems (e.g., based on sensor networks). Usually, these applications do not require system-provided fault tolerance, but they greatly benefit from the lower build complexity. Still, they may need to compensate for jitter in the order of message periods during message reception by appropriate buffering or queuing. In the case of monitoring systems, observations can be time-stamped by reading the global time available at the LIFSS after the observation. Applications that use time-synchronized interfaces trade the added build complexity for the realization of deterministic real-time transactions with short end-to-end delays. Typical examples of such applications are related to control tasks (e.g., flight control systems, film stretching systems), which often require highly regular sense-control-act cycles. Some control systems are safety critical and profit from fault tolerance that is enabled at the architectural level by the use of time-synchronized interfaces.

6 Example: CC Predictability

In this section we show the temporal predictability of components using the different access strategies to the interface of a time-triggered communication system. Figure 9 shows the read/write behaviour at a local network node between the CC and the LIFSS:

- The first row shows the time line of the write access to the LIFSS, with different message instances over time.

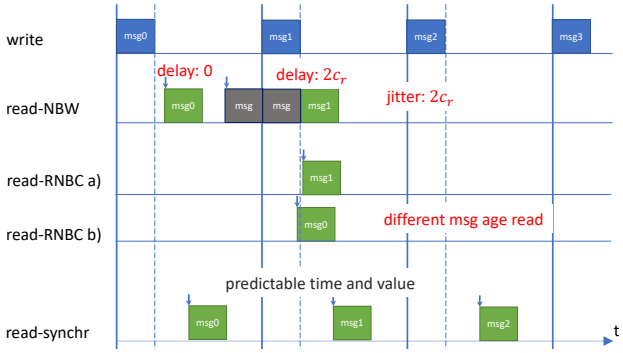

Figure 9. Local behaviour of different interfacing methods

- The second row "read-NBW" shows examples of asynchronous read access via the NBW protocol. The first read operation happens just after completion of the last write access, and proceeds with zero delay. The second read operation just slightly overlaps with the write operation for message *msg1*. This lead to two read attempts partially overlapped with the write, causing them to fail, and only the third attempt then finally succeds. Thus, NBW causes jitter to the temporal behaviour of a component.

- The rows "read-RNBC a)" and "read-RNBC b)" show examples of the asynchronous read with RNBC. In both cases, the read operation has a constant execution time, as RNBC is free of read-/write conflicts by design. However, what the two different examples also show, is that with RNBC a small difference in the start of a read can cause to read a different message instance. Thus, while RNBC allows components to be time predictable, the predictability in the value domain of messages is not given.

- Finally, the row "read-synchr" shows access patterns with synchronous interfacing. It shows that synchronous interfacing not only gives constant execution time of the read operation, but also provides predictability in the value domain, as read operations are fixed scheduled with some time offset after the completion of each write access.

7 Conclusion

In this paper we discussed different interfacing techniques between a computing component (CC) and a time-triggered communication network. The results show that best predictability in time and value domain, and also the fault tolerance needed for safetey-critical applications is achieved by synchronous interfacing, where the execution of activities at the CC is synchronous with the message communication over the time-triggered communication system (TTCS). To achieve this, the local clock of the CC has to be synchronized to the global clock of the time-triggered network. With asynchronous access, neither clock synchronisation nor global scheduling of activities is required, but it comes at the cost of unpredictability in the value domain, as it is not ensured which message instance is obtained by a read access. We have also developed a new asynchronous access protocol, called *Rate-bounded Non-Blocking Communication* protocol (RNBC), which ensures the temporal predictability and autonomy at the component level for asynchronous access. For the case that the timing requirements cannot be met with RNBC, we have also extended RNBC to a multi-buffer variant, which can be made schedulable for any timing requirements by adjusting the buffer size.

References

[1] J. R. Engdahl and D. Chung. Lock-free data structure for multi-core processors. In *Proc. Int'l Conference on Control, Automation and Systems (ICCAS 2010)*, pages 984–989, Oct 2010.

[2] H. Kopetz. Event-triggered versus time-triggered real-time systems. In A. Karshmer and J. Nehmer, editors, *Proc. Int. Workshop on Operating Systems of the 90s and Beyond*, Lecture Notes in Computer Science, Volume 563, pages 87–101, Berlin, Germany, 1991. Springer.

[3] H. Kopetz. The time-triggered model of computation. In *Proc. IEEE Real-Time Systems Symposium*, pages 168–177. IEEE, 1998.

[4] H. Kopetz. *Real-Time Systems*. Springer, 2nd edition, 2011.

[5] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[6] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. In *Proc. 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 310–315. IEEE, 1997.

[7] H. Kopetz and R. Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, 13(4):156–162, 2002.

Accepted Manuscript.
Article accepted for publication in the Journal of Systems Architecture, 22/12/2019.
https://doi.org/10.1016/j.sysarc.2019.101690

[8] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Proc. IEEE Real-Time Systems Symposium*, pages 131–137, Dec. 1993.

[9] H. Ma and Y. Li. Effective data exchange in parallel computing. In *Proc. Int'l Conference on Information Science and Cloud Computing*, pages 106–112, Dec. 2013.

[10] S. Maurer and R. Kirner. Cross-criticality interfaces for cyber-physical systems. In *Proc. 1st IEEE Int'l Conference on Event-based Control, Communication, and Signal Processing*, Krakow, Poland, June 2015.

[11] J. S. McVeigh, V. S. Grinberg, and M. Siegel. Double-buffering technique for binocular imaging in a window. In *Proc. SPIE 2409, Stereoscopic Displays and Virtual Reality Systems II*, Mar. 1995.

[12] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1995.

[13] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transaction on Compututers*, 49(2):100–111, Feb. 2000.

[14] P. Puschner and R. Kirner. From time-triggered to time-deterministic real-time systems. In *Proc. 5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 115–124, Braga, Portugal, Oct. 2006.

[15] R. B. Sheeparamatti, B. G. Sheeparamatti, M. Bharamagoudar, and N. Ambali. Simulink model for double buffering. In *Proc. 32nd Annual Conference on IEEE Industrial Electronics (IECON'06)*, pages 4593–4597, Nov. 2006.

[16] H. A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, MA, USA, 1996.

[17] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), Apr. 2008.