CarRing IV- Real-time Computer Network

Doctoral Thesis
(Dissertation)

to be awarded the degree
Doctor of Engineering (Dr.-Ing.)

submitted by
M.Eng. Ahmad Obeid
from Baghdad, Iraq

approved by the Faculty of Mathematics/Computer Science and Mechanical Engineering
Clausthal University of Technology

Date of oral examination
24 May 2022

Dean
Prof. Dr. rer. nat. Jörg P. Müller

Chairperson of the Board of Examiners
Prof. Dr.-Ing. Volker Wesling

Supervising tutor
Prof. Dr. rer. nat. Christian Siemers

Reviewer
Prof. Dr. rer. nat. Rüdiger Ehlers

CarRing IV- Real-time Computer Network

M.Eng. Ahmad Obeid

Abstract

Whether be it automotive, avionics or automation, advances in their respective real-time communication technology focus on further improving preexisting solutions. For in-vehicle communication, the ever-increasing number of computer-based systems, applications and connections as well as the use of multiple proprietary communication standards results in an increasingly complex wiring harness. This is in-part due to those standards being incompatible with one another. In addition to cost, this also impacts weight, which in turn affects fuel consumption.

The work presented in this thesis is in-part theoretical and in-part applied. The former is represented by a new protocol, while the latter corresponds to the protocol's hardware implementation. In the first part of the thesis, the real-time communication protocol of CarRing IV (CR-IV) is presented. It provides isochronous and hard real-time guarantees without requiring network-wide clock synchronization. With up to 16 nodes per ring, a CR-IV network can consist of as many as 256 rings interconnected by routers. CR-IV uses a reduced OSI model (layers 1-3, 7), which is both typical of and preferable for its application areas. Moreover, it supports both event- and time-triggered communication paradigms. The transparent mode feature allows CR-IV to act as a backbone for existing networks, thereby addressing incompatibility concerns and easing the transition into a more unified network solution. Using this feature, user devices can communicate with one another via a CR-IV network without requiring user interference, or any user device or application changes. Combined with the protocol's reliable multicast, the feature extends CR-IV's capabilities to include field bus emulation. The second part of the thesis presents the other important aspect of CR-IV. All of its OSI model layers are implemented in an FPGA using Hardware Description Languages (HDLs) without relying-on or including any hard or soft processors. CR-IV's Register-Transfer Level (RTL) hardware design is created using a new approach that can best be described as token-based data-flow. The approach is both vertically and horizontally scalable. It uses stateless and loosely coupled Processing Elements (PEs) as well as arbiter/memory allocation pairs. By having granular control and compartmentalizing every aspect of a solution, the approach lends itself to being used for implementing other software-level solutions in hardware.

Many test scenarios are conducted to both highlight and examine the results achieved in CR-IV. Those scenarios range from direct performance measurements to behavior-specific tests. Moreover, a lab-demo is created that essentially amounts to a proof of concept. The demo represents a practical test as opposed to a scenario-specific one. Whether be it test scenarios or the lab-demo, all are carried-out using the project's prototype boards, i.e. no simulation tests. The results obtained represent CR-IV's real-world realistic outcomes with up to 13.61 Gbps.

Acknowledgments

My sincere, earnest and deepest gratitude to Professor Christian Siemers without whom this would not have been possible. I feel incredibly fortunate, especially in these difficult times. Christian's helpful advice, encouragement, guidance and support were indispensable. Few people have been pivotal in my life trajectory, Christian is absolutely one of them. Hopefully, I would someday be able to pay-it-forward.

Thanks to Professor Harald Richter for the employment opportunity through-which I was able to support my parents and sisters.

Many thanks to my second supervisor Professor Rüdiger Ehlers, chairman Professor Volker Wesling and my thesis committee members.

I am greatly thankful to Florian Pramme, Alexander Keidel and Christian Marg for their friendship, support and sharing the ups and downs with empathy and many many moments of levity. And I would like to also thank Peter Platzdasch, Christine Kammann, Thomas Bravin, Sandra Karpenstein and Professor Jörg Müller.

Finally, this thesis is dedicated to the most important people in my life, my family. Their supportive and loving environment helped immensely.

Table of Contents

List of Publications

A. Obeid and H. Richter. Routing in CarRing 4 - a transparent communication mean for field buses and LANs. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–8, 2017. URL: https://doi.org/10.1109/CCWC.2017.7868396.

H. Richter and A. Obeid. Architecture and performance of CR4 - a transparent communication mean for field buses and LANs. In *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 53–60, 2015. URL: https://doi.org/10.1109/ICUMT.2015.7382405.

H. Richter and A. Obeid and M. Glukhikh and M. Moiseev. Layer 1 and 2 of a ring-based, real-time network for in-vehicle communication. In *2014 6th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 123–130, 2014. URL: https://doi.org/10.1109/ICUMT.2014.7002090.

List of Figures

Chapter 1

Introduction

This chapter introduces the background for the primary topic of this thesis, which is hard real-time communication. Then, a description of the problem being addressed in this thesis is provided, and followed by thesis contributions. Finally, the scope of this thesis and its organization is described.

1.1 Background

Real-time spans multiple topics in computer science and engineering ranging from high performance computing, Operating Systems (OSs), embedded systems to computer networks. For example:

- The authors in [24] use the processing time of the used algorithms, the computation time of the used Central Processing Units (CPUs) and Graphical Processing Units (GPUs) as well as the frame rate of their architecture measured in frames per second for their real-time high performance computation. They combine Cloud technology with GPUs within the framework of real-time high performance computation architecture for the application of detecting and tracking multiple moving targets based on Wide Area Motion Imagery (WAMI).

- Real-time OSs are typically used when designing a real-time embedded system, e.g. FreeRTOS [34] and uC/OS-III [28]. An example of those real-time embed-

ded systems can be found within automobile control systems and production line control systems. Such control systems are inherently distributed, where the various system components (e.g. sensors and actuators) are placed at different physical locations. Therefore, they rely on an underlying real-time network(s) to maintain the system's real-time behavior.

In addition to the wide spectrum of application areas, guaranteeing and maintaining real-time behavior is achieved using a range of different approaches. Those are covered in sub-section 3.1.1 with respect to real-time communication. Despite this wide range of applications and approaches, there are common core definitions and classifications.

Real-time is defined as a quantitative notion of time and is measured using a physical (real) clock [25]. Real-time systems are systems that need a quantitative expression of [*maximum reaction*] time to describe their behavior [25]. Those systems and their applications are commonly classified as soft, firm or hard real-time [25]. Within the latter, there is also weakly hard real-time [13] and isochronous real-time.

In soft real-time, failure to satisfy timing constraints does not put the system in a failed state, rather it causes a degradation in the utility of a task's result or transmitted data. Similarly, no system failure occurs in firm real-time. However, the task's result or transmitted data is not utilized, i.e. rejected.

On the other hand, in hard real-time, a failure to satisfy timing constraints does put the system in a failed state. As for weakly hard real-time, it corresponds to a system in which the distribution of its met and missed deadlines during a window of time is precisely bounded [13]. Last but not least, isochronous real-time can be defined as delivering data at a specific time instant, which is more strict than having a hard upper time limit, i.e. hard real-time.

In this thesis, the problem being addressed and the contributions made, fall into the category of real-time communications and under the classifications of isochronous and hard real-time.

1.2 Problem Description

Whether due to driver assistance, safety, infotainment or various x-by-wire applications, in-vehicle networks and communications are growing ever-more complex. The increased complexity is also driven by the ever-increasing number of computer-based systems, applications and connections. This in-part results in adding more sensors and Electronic Control Units (ECUs) as well as increased communication requirements. For example, the latter is a by-product of the introduction of more advanced driver assistance systems, e.g. camera-based solutions.

Furthermore, in-vehicle networks are divided internally into separate branches or domains interconnected by gateway(s). Those domains use multiple different proprietary communication standards that are incompatible with one another. Those standards also use different network interfaces and cabling. This approach to in-vehicle networks not only results in a more complex wiring harness, but also drives-up the cost of cabling and maintenance. Moreover, it also increases the vehicle's weight, which negatively impacts fuel consumption.

Thus far, advances within in-vehicle networks have focused predominantly on further improving preexisting solutions, rather than reducing the number of standards, i.e., moving towards a more unified solution. Such a solution should ideally be capable of complying with strict timing requirements, have high bandwidth as well as be able to act as a backbone for preexisting solutions, thereby easing their transition to said solution. Another equally important aspect of such a solution is its implementation approach.

Regardless of the communication protocol, higher layers are typically realized by means of a software implementation and lower layers are implemented in hardware. The software implementation is composed of the higher protocol layers and a device driver that is used to communicate with the hardware side, e.g. a Network Interface Card (NIC).

As for the hardware implementation, it is typically composed of a PHYsical interface transceiver (PHY), Media Access Control (MAC), an Application-Specific Inte-

grated Circuit (ASIC), memory and an interface to the system's data bus. The ASIC acts as a control unit that handles interrupts and manages the flow of data, which includes the transmission data buffered in memory. The transfer of data to and from the network is handled by MAC and PHY. Those two components implement the data link and the physical layers of the Open Systems Interconnection (OSI) model [21], respectively.

Such a software-hardware implementation approach of a communication protocol suffers from the implicit overhead caused by connecting the two, e.g., device drivers and intermediate interfaces. Furthermore, real-time behavior, especially one with more strict timing requirements, is typically maintained by the higher protocol layers. This necessitates the inclusion of those layers and consequently the associated overhead.

Converting the aforementioned software implementation of higher protocol layers into hardware can not be easily done without inefficiencies. There is currently no standard approach of implementing software-level solutions in hardware. However, there are software tools that provide guided steps to convert an existing software code to hardware design code, which can result in an inefficient implementation. Other tools focus on accelerating the software solution by offloading parts of the software code to hardware in hope of achieving faster execution time.

1.3 Thesis Contributions

The contributions of this thesis are mainly in two parts, a new communication protocol and hardware implementation approach.

First contribution: new protocol

The CarRing IV communication protocol supports both isochronous and hard real-time as well as event- and time-triggered communication paradigms. The latter is primarily based on the protocol's `SendAt` feature, which allows for periodical and delayed data transmissions.

The provided real-time guarantees do not require network-wide clock synchronization. Nevertheless, synchronization is included and it uses a new decentralized approach, where the main reference node is only responsible for synchronizing the nodes within its own ring as opposed to the entire network.

The backbone capability, mentioned in the previous section, is made possible by the protocol's transparent mode feature. It transparently transfers data between user devices, wherein multiple devices can be connected per network node. No modification to the device or user interference is required, rather the device can be directly connected and proceed to transmit data. Combined with the protocol's reliable multicast, the transparent mode feature extends CarRing IV's capabilities to include field bus emulation, where Controller Area Network (CAN) is regarded as the primary use-case.

Second contribution: the protocol's hardware implementation

All OSI model layers of CarRing IV's communication protocol are implemented in hardware and more precisely in a Field-Programmable Gate Array (FPGA) using Hardware Description Languages (HDLs). This allows CarRing IV to be as close as possible to wire-speed.

Not only is the protocol implemented in hardware, but the implementation is achieved using a new approach. It can be best described as token-based data-flow. It is inspired by the data-flow model in that process execution is driven by the availability of a token. However, this token is not used to transfer data between processes, rather it is used to transfer the state of the data being processed. It is also inspired by control-flow model, in that shared memory is used between layers for the data being processed and access to this shared memory is managed by the aforementioned tokens.

The approach is both vertically and horizontally scalable. It does not include any hard- or soft-processors, i.e. no CPUs. Rather, it uses stateless and loosely coupled Processing Elements (PEs) as well as arbiter/memory allocation pairs. Each pair corresponds to a frame being processed. The arbiters manage memory access and execution order between PEs.

By having granular control and compartmentalizing every aspect of a solution, the approach can also be used to efficiently implement other software-level solutions in hardware, where the design and implementation presented in this thesis can be referenced as an example implementation.

Previous work and other contributions

As the title implies, the work presented in this thesis represents the fourth result of a long-term project, called CarRing. As stated in [20], the overarching goal of CarRing is to establish a real-time computer network that has high reliability, predictable time-behavior, good scalability from small to large systems, high flexibility, high interoperability, high data rates, low delay and affordable cable lengths. In practical terms, the project aims at providing both a theoretical result, i.e. a communication protocol, and a tangible result, i.e. a prototype.

In what follows, brief descriptions of other contributions made by me and other participants in this long-term project are provided. In addition to contextualizing the work presented in this thesis, they also include previous protocol-specific contributions.

The first result, i.e. CarRing I, of the long-term project is a network simulation and the second result is based on ten Microblaze soft-core processors. Microblaze is a full-featured and FPGA-optimized 32-bit Reduced Instruction Set Computer (RISC) soft-core processor created by Xilinx. The second result used the software-hardware implementation approach described in the previous section.

Although the fourth result is presented in this thesis, my work and contributions started with the CarRing project's third iteration. Within the context of CarRing III, I was a co-contributor, where I have made major and substantial contributions. While CarRing III is outside the scope of this thesis, my contributions included, but were not limited to, application and network layers as well as router design, the SendAt and transparent mode features. The later two are described in sub-sections 5.2.6 and 5.4.6, respectively.

Due to being the first hardware-only implementation of the protocol, CarRing III

had only few commonalities with its predecessors. The implementation relied heavily on First-In-First-Out (FIFO)-buffers as the main design component for the protocol layers. In other words, the third result is in-part a FIFO-based implementation of the CarRing communication protocol.

During the CarRing III and CarRing IV projects' life-time, two prototypes were created by three contributors (Me, Project Manager Prof. H. Richter, Third-party manufacturer). Throughout the thesis, those prototypes are referred to as Virtex5- and Kintex7-prototype, i.e. named after their respective FPGA device families. While a detailed description is outside the scope of this thesis, those prototypes are nevertheless briefly described in multiple chapters based on context.

There is also a fifth result of the long-term project. However, the sole purpose of CarRing V is the development of the Kintex7-prototype. In other words, the fifth result does not affect the communication protocol, i.e. the long-term project is finalized with the work presented in this thesis. As mentioned earlier, I am a co-creator of the latest prototype.

Last but not least, prior to the work presented in this thesis, the most recent contribution to the communication protocol was made in CarRing II by M. Wille. In his thesis, a six layer communication protocol is described along with an exemplary implementation, which relies on the aforementioned soft-core processors.

In contrast to M. Wille's version, the CarRing IV communication protocol differ drastically in all areas, i.e. functionality, design and implementation. Comparing and contrasting both protocol versions, in all the aforementioned areas, is outside the scope of this thesis. Nevertheless, in what follows, a brief description of the main differences is provided.

The former version is a six layer protocol starting with layer 2 up to 7, as opposed to the four layer version (1-3,7) presented in this thesis. As for the protocols' functionality, few basic concepts and some carry-over keywords are preserved in the new version for continuity reasons, e.g. names of the protocol operation modes. However, the inner-workings are entirely changed in most cases and drastically modified in some others. Moreover, the former version is implemented using C-language and

the resulting instructions are executed in-part using hard and soft-processors, i.e. software-level solution. As such, the valuable contributions of M. Wille are regarded as a feasibility study that helped inform the development of the new protocol.

1.4 Challenges

As stated in the previous section, thesis contributions are in-part theoretical and in-part applied. Combined with the Register-Transfer Level (RTL) implementations, realizing the CarRing IV project goals presented significant challenges. To focus on those challenges and maintain brevity, detailed descriptions of the topics highlighted in this section, are delegated to subsequent chapters within the thesis.

The most important aspect of a real-time protocol is providing timing guarantees. Within the context of CarRing IV, this is further complicated by the protocol's implementation. RTL designs have their own timing constraints, which are separate and apart from the protocol's timing behavior. Abiding by those timing constraints is crucial not only for the protocol's functionality, but also for its timing guarantees. In other words, the timing behavior of both the design and the protocol must be maintained. For example, abiding by the design's timing constraints could result in changes to the logic operations associated with the protocol's functionality. Such changes must either be negligible or have no impact on the protocol's delay.

The designs are complicated even further by one of the protocol's features, called transparent mode. By definition, the feature requires the protocol's application layer to control and transfer data via multiple other interfaces within the project's prototype, e.g. Ethernet, CAN. Those additional interfaces represent different networks, i.e. different protocols and transceivers. This introduces multiple other non-CarRing-protocol specific designs that must be separately developed and tested. Furthermore, it shifts the main design from a single-clock to a multi-clock domain design, where meta-stability is a more present and urgent issue.

Other than implementation, Protocol Data Unit (PDU) sizes vary between the different protocols. As such, possible unfair delays must be accounted-for and fairness

must be maintained when transmitting those differently sized PDUs.

Another challenge is the RTL implementation of higher protocol layers, which are typically realized on the software-level. Even if pre-existing software-level approaches were to be adapted, this would result in multiple new sub-projects. In other words, the convenience of relying on existing frameworks or libraries, while typical on the software-level, is practically non-existent for hardware-level implementations.

Although freely accessible libraries do exist and some are used within CarRing IV, they are of no relevance to the issue at hand, i.e. implementing software-level solutions in hardware. Rather, they are either specific to a design component or cover rudimentary logic operations. As for the commercial counterpart, they are associated-with and specific-to the manufacturer's product or product-line, i.e. either associated with specific chips or a device family.

Last but not least, the combined complexities of the designs, hardware components as well as the used software tool-chain and its compute intensive operations resulted in a relatively limited amount of tests that can be carried out on the project's prototype. It was at most 2 to 3 tests per work day. Furthermore, the development of both the protocol and prototypes were carried out simultaneously. This necessitated the introduction of multiple verification steps in addition to those typically associated with FPGAs and RTL designs, thereby lengthening the development process.

Note that the complexities introduced by the tool-chain's software applications were in-part due to the underlying FPGAs and chips. In other words, the complexities are in-part inherent to the devices at hand, rather than being completely due to the fact that those applications were developed by different companies, or the quality of the developed applications.

1.5 Thesis Scope and Organization

The thesis focuses on two main topics, which are hard real-time communication protocols and hardware implementation of software-level solutions. However, the scope of this thesis is not limited to the aforementioned topics.

During the development process, several simulation and hardware tests were performed. Among those hardware tests, there are additional tests that incorporated multiple different interfaces from other existing networks, e.g. Ethernet and CAN. The purpose of those additional tests is to provide a tangible proof that CarRing can be used as a backbone for existing networks, which is one of the use-cases for the protocol's transparent mode feature.

For that purpose a lab demo was created in which a car headlight, two laptops, a network camera and two motors were connected to and communicated with each other via a CarRing network. Consequently, this extended the scope of the work done in this thesis to include other interface-specific hardware designs and components. Furthermore, other hardware implementation challenges were also introduced, e.g. Multi-Clock domain designs.

The rest of this thesis is organized as follows. It starts by describing the preliminaries, in chapter 2, required for the rest of the thesis. This chapter includes the used languages, tools and technologies as well as their respective specific terms and keywords.

Although, to the best of our knowledge, there is no existing solution that can be used for direct comparison, chapter 3 discusses related work.

The big picture is presented in chapter 4. From the point of view of a user, it describes how the solution's various parts are used and how they interact to provide the expected functionality.

After discussing related work and describing the functionality of the solution as a whole, a description of the underlying layers that form the solution is provided. Chapter 5 starts with the bottom layer all the way up to the user interface. This chapter also describes the aforementioned feature, i.e. transparent mode.

Some key parts of the protocol layers require their own dedicated chapters. Chapter 6 presents a non-classic approach to achieving a network-wide clock synchronization, which is aimed at time-triggered real-time data transmissions. And chapter 7 presents the solution's reliable multicast approach, which can also be used for bus emulation.

Chapter 8 describes the second topic of this thesis, which is the RTL hardware design and implementation of the solution.

After covering the thesis two main topics, the results and measurements are presented in chapter 9. Finally, chapter 10 summarizes the thesis and briefly describes future work.

Chapter 2

Preliminaries

The work presented in this thesis is in-part theoretical i.e. the fourth version of the CarRing communication protocol, and in-part applied i.e. the protocol implementation and additional protocol-related implementations. The applied part uses preexisting languages, software tools and technologies. Those include their own sets of terms, which are used throughout the thesis. This chapter briefly explains those languages, software tools and technologies as well as their terms.

2.1 System Design and Hardware Description Languages

Three languages are used to design and implement the CarRing IV communication protocol and its features. Two languages, SystemC and VHSIC Hardware Description Language (VHDL), are explicitly used, while Verilog is used implicitly. In what follows, a brief description of those languages is given as well as the associated terms and the context in which they were used.

2.1.1 VHDL

VHDL is a hardware description language standardized by IEEE [2]. It contains language constructs that are used to describe hardware digital designs. The language

allows for hierarchical designs, and it supports both top-down and bottom-up design approaches. From the top-level perspective, the CarRing IV design as a whole, is implemented using VHDL. The design consists of various sub-designs that are implemented using both VHDL and Verilog. Those sub-designs are covered in chapter 5. The VHDL-specific sub-designs are mainly concerned with the control logic for the various types of CarRing IV supported interfaces and their corresponding transceivers in the CarRing IV prototype board, which are described in section 2.3. Managing and handling those interfaces, is essential for one of the CarRing IV features, called transparent mode, which is explained in sub-section 5.4.6.

There are two basic description styles in VHDL, `behavioral` and `structural`. A structural description is essentially a schematic, representing a block diagram or a circuit diagram [16]. Rather than the actual circuit, a behavioral description is more abstract and focuses on describing the behavior of the design, which includes the use of language constructs that resemble sequential semantics. Those semantics include the use of variables and sequential execution, which are encapsulated in a VHDL `process` [16]. Such a process is invoked, i.e. runs, when there is a change detected in any `signal` in its `sensitivity list`. A VHDL signal is the equivalent of a "wire" and it represents a communication channel among processes or sub-designs within a design. As the name suggests, a sensitivity list is a list of signals that a process is sensitive to any change in any of its signals. Each VHDL process has its own sensitivity list.

Regardless of the description style, whether the design is a set of processes or a set of interconnected components or a combination of both, the basic building block of a design is called an `entity`. An entity has two main parts, `port` and `architecture`. Ports represent the entity's external interface, i.e. the communication channel with other entities or design components. However, ports of the top-level entity are connected to the physical pins in the actual hardware. The software tool that makes the connection possible is explained in section 2.2. As for the architecture, it is a language construct that encapsulates the aforementioned descriptions. An entity can have multiple architectures, where one can be a structural and another can

be a behavioral description of the design.

Another noteworthy feature of VHDL, that was used in CarRing IV, is the concept of a `package`. It is a language construct that groups a variety of declarations, e.g. user defined data types, which can then be shared between several entities.

2.1.2 Verilog

Verilog is another hardware description language standardized by IEEE [1]. Similar to VHDL, it also contains language constructs that are used to describe hardware digital designs. In the CarRing IV design, the use of Verilog is implicit. One of the used software tools, accepts as an input a SystemC design and outputs a Verilog version of the SystemC design. This software tool is called C-TO-Silicon (CTOS) and it is explained in section 2.2. Although there were no designs written explicitly using Verilog, knowledge of the language was still required for simulation tests and debugging purposes.
Verilog is quite similar to VHDL. Therefore, rather than partially repeating the previous section, in what follows, VHDL equivalencies and contrasts are provided.

Similar to VHDL, Verilog also supports both description styles, behavioral and structural. The Verilog equivalent of a VHDL entity is a `Verilog module`. Although modules also have ports, they lack the language construct equivalent of an architecture. As a result, a module is restricted to one description of the design. In Verilog, `wire` is the functional equivalent of signal in VHDL. A process is represented by an `always` statement and it too has a sensitivity list.

The use of VHDL instead of Verilog, was an early design decision. Despite the advantage of reducing the number languages used in the project, we have continued to use VHDL due to its notable advantages in behavioral modeling and design reusability. More specifically, the support for user defined data types, which is very helpful in behavioral description of a design. Also, there is no equivalent concept in Verilog for a VHDL package. Last but not least, VHDL is a strongly typed language, which gives the advantage of reducing potential modeling errors. For example, Verilog allows automatic padding of signals, when there is a difference in bit widths.

2.1.3 SystemC

SystemC is a system design and modeling language [14]. It is based on a well established language, C++, and is on a higher level of abstraction than VHDL and Verilog. It provides a set of modeling constructs that are similar to those used for RTL and behavioral modeling within VHDL and Verilog [36].

Although it is meant primarily for system design and modeling, SystemC is used to implement the CarRing IV communication protocol. This is made possible using a software tool called CTOS, which is described in section 2.2.

CarRing IV protocol's SystemC design is composed of interconnected design components called `SystemC modules`. They are similar to VHDL entities or Verilog modules. A SystemC module or `SC_MODULE` is the smallest container of functionality with state, behavior and structure for hierarchical connectivity [14]. SystemC modules are C++ classes that can be interconnected using `SystemC channels`, which are also C++ classes. SystemC channels serve as a container for communication functionality. They provide a higher layer of abstraction by encapsulating communication details between design components. Modules use `SystemC ports` to connect to one or more channels. Ports are represented as class members within a module and each port is basically a C++ pointer to a channel. Once ports are set, the channel's class methods are used by the corresponding modules to communicate with each other. Although those methods are implemented by the channel, they are inherited from a separate abstract class called `SystemC interface`. Ports connect to channels through interfaces.

Within the context of SystemC channels, an additional CTOS library[15] is used in this project. The `Flex Channel` library contains a set of highly reusable basic blocks for point-to-point communication. It relies on the use of `Transaction Level Modeling (TLM)` put and get interfaces to send and receive data through the channel.

TLM is a modeling concept that is language independent. In this thesis, TLM is used as captured by the SystemC language. At the time of writing this thesis, there are two versions of SystemC TLM. The first version includes core interfaces,

TLM_FIFO and analysis interface and port. The second version includes the first and adds new utilities and an interoperability layer that, in turn, includes new features and interfaces. In this thesis, the SystemC TLM used is CTOS TLM, which is a subset of the first version of SystemC TLM.

Essentially, TLM-based modeling allows for early testing and functional verification, which has significant improvements on productivity. In the early phase of the CarRing IV design, the development efforts were focused on the design of the protocol layers and their components, rather than the communication channels between them. Using TLM approach, the communication channels were abstracted using TLM interfaces. As development progresses, the timing accuracy of the communication channels was increased from un-timed to approximately timed and then to accurately timed without affecting the protocol layers and their design components. In the case of un-timed, it was direct C++ software code within the communication channels. As for approximately timed, those same software code segments were developed further to include artificial delays. The use of Flex Channels in CarRing IV design, is an example of an accurately timed communication channel.

Last but not least `SystemC processes`, they are the basic unit of execution in SystemC [14]. They are represented as class methods of a module and use the module's ports to access external channels. They have two main types, `SC_METHOD` and `SC_THREAD`. An `SC_METHOD` is a SystemC process that executes without interruption, i.e. it cannot be suspended. On the other hand, `SC_THREAD` can be suspended. The suspension is performed by calling the wait() function within the process. In case of `SC_THREAD`, execution is resumed when one of the signals in the process's sensitivity list changes. A process sensitivity list in SystemC is similar to that of VHDL and Verilog. `SC_CTHREAD` or clocked thread process, is another process type and can be seen as a derivative of the `SC_THREAD` process type. It is the main process type used in the SystemC design of the CarRing IV protocol. Its sensitivity list consists of only one signal, called clock. In SystemC, a clock is a signal that emulates the hardware behavior of a single-ended clock which is explained in section 2.3.

2.2 Tool Chain

During the development phase of the project, multiple software tools are used to create, test and implement VHDL and SystemC designs. Those tools include an Integrated Development Environment (IDE) and Electronic Design Automation (EDA) software tools. The EDA tools are explained in the following sub-sections. As for the IDE software tool, Microsoft Visual C++ (MSVC) is used . Since SystemC is C++-based, a SystemC simulation test is essentially a software application created by a C++ compiler which is in this case, MSVC. Thus, each new test entails compiling and running a new executable. And the test results are either directly displayed or logged to a file.

2.2.1 Cadence C-to-Silicon

CTOS is a software tool that allows a design engineer to generate a functionally equivalent RTL Verilog design from a higher-abstraction SystemC design. In this project, it is used to implement SystemC designs. This implementation is a multi-step process, where the input is a SystemC design and the output is a functionally equivalent Verilog version of the design. Those steps include preparing a SystemC design, specifying micro-architecture, scheduling, analysis and implementation.

Preparing a SystemC design

The term preparation refers to relatively few and minor adjustments made to a SystemC design. The aim is to meet coding constraints, which are defined by CTOS. This step consists of two sub-steps. First, limiting the use of SystemC language to the `synthesizable` subset of SystemC, also known as SystemC RTL. Second, abiding by CTOS restrictions on coding styles and SystemC. Both sub-steps are detailed in the CTOS user guide [15]. In what follows, a brief explanation of the term synthesizable is provided.

In SystemC, designs can be written using all the features offered by the language. Those features include debugging, signal tracing, error and message reporting. Such

features can not be directly represented by real hardware. For example, bit-wise operators can be represented by gates, whereas a SystemC function that logs signal changes to a file, can not be directly represented by a hardware element. If such features are used in a SystemC design, then that design is not synthesizable.

Specifying micro-architecture of a design

It is a step where the implementation of the design's micro-architecture must be resolved. In this context, the design's micro-architecture refers to the design's functions, loops and arrays. And the term resolved refers to the way those three are represented/implemented. For example, for-loops do not have a direct synthesizable HDL construct, i.e. some form of a hardware component. As such, they must be resolved into a synthesizable design, e.g. a loop can be unrolled. In this case, the loop is represented as a replicated set of statements, where each set refers to one loop iteration.

CTOS provides multiple resolution techniques. They depend in-part on the capabilities of the underlying hardware. Furthermore, they affect the area and timing of the SystemC design. In other words, they affect the required amount of FPGA resources and the consumed number of clock cycles, i.e. latency of design. Explaining each resolution technique is beyond the scope of the thesis. Nevertheless, those techniques are briefly mentioned for completeness. Also, some of them are self-explanatory.

Resolving a function can be performed by inlining, pipelining, conversion to a Look-Up Table (LUT), or importing an RTL IP. And Loops can be resolved via unrolling, breaking or pipelining. As for arrays, they are treated as memory components of a design. As such, they can be implemented using registers or Random Access Memory (RAM). Resolving arrays can be performed by flattening, merging, splitting, restructuring or allocating memory and RTL IP, as well as floating Input/Output (I/O) and array accesses.

Scheduling a design

It is the most time-consuming and critical step. If successful, the output of this step is the aforementioned RTL Verilog design, i.e. a synthesizable design that can be used in a Xilinx ISE or Vivado project. In short, the CTOS scheduler maps operations to resources and thereafter bind those resources to states [15]. The term operations refers to statements written within a `SystemC process`, e.g. add operation or read/write to memory component. And the term resources refers to hardware representation/implementation of those operations. For example, the operation of multiplying two values is mapped to a multiplier component, wherein the bit-width of the input/output ports is specified.

In essence, the term states refers to the mechanism with which CTOS ensures the order of operations within a `SystemC process`. For example, they could be created as a by-product of managing the order of read/write operations from/to a memory component, or as a by-product of managing operations within a pipelined loop. More commonly, they are created due to `wait()` function calls, which is described in sub-section 2.1.3. With respect to the scheduled output, i.e. the RTL Verilog design, those states are represented by multiple Finite-State Machines (FSMs).

Analysis and Implementation

Depending on the output of the CTOS scheduler, this step can be reduced to generating the RTL Verilog design files. It revolves around improving timing and area of the resulting design as well as design verification. Within the context of timing-related improvement, a timing report is generated wherein possible timing requirement violations can be identified. For example, negative slacks can be identified and resolved based on the context in which they occurred, i.e. the corresponding `SystemC process` and related operations. The term negative slack is described in more detail within sub-section 2.3.3.

Resolving negative slacks is also aided by another CTOS feature called Cycle Analysis. It allows for a more granular check. Each cycle, in which a negative slack occurs,

can be individually examined. Those cycles are depicted using diagrams that show the operations involved and the amount of time each operation consumes within the clock cycle.

As for area-related improvement, this refers to measures that could be applied to reduce the overall area, i.e. FPGA resources required, by the design. For example, reducing size of registers used, resource sharing, or reconsidering resolution techniques used in the micro-architecture step e.g. flattening arrays.

2.2.2 Xilinx ISE and Vivado Design Suites

Despite serving the same purpose, two software tools are used in CarRing IV to convert RTL-level designs into an FPGA-specific configuration file. In essence, this conversion consists of three main steps, i.e. logic synthesis, implementation and bitstream generation. In turn, each step consists of multiple sub-steps. Providing a detailed coverage of all (sub-)steps is beyond the scope of this thesis. Nevertheless, in what follows, a brief description is provided for each step.

As for the necessity of using both tools, it stems from their respective list of supported FPGAs and CarRing IV's prototype boards. Since Vivado does not support Virtex-5 FPGAs, Xilinx ISE is used for the previous Virtex5-prototype, whereas Vivado is used for the current Kintex7-prototype. FPGAs are covered in more details in subsection 2.3.1.

Synthesis

It is a refinement process that realizes a netlist from an RTL design. Three HDLs can be used for the input RTL design, namely VHDL, Verilog and SystemVerilog. The term netlist corresponds to the structural view of a design, wherein the internal implementation is described. It is list of nets, where each net corresponds to a set of wire connections and its respective component. As such, the list describes which components are used and how they are interconnected. Thus, it can be considered as a circuit description or the design's schematic.

As the refinement process progresses, multiple netlists are generated i.e. RT-, gate- and cell-level netlists. Within each level, the corresponding circuit is constructed using components from the level's respective library. Examples of RT-level components are arithmetic operators, multiplexers, registers. Using such components, the circuit description, i.e. RT-level netlist, is generated from the input RTL behavioral description.

As the name implies, a gate-level netlist is generated using gate-level components. Both RT- and gate-level netlists are technology independent, i.e. generated using generic components that are not specific to the underlying FPGA. On the other hand, cell-level netlist is generated by essentially mapping the generic gate-level components into logic cells of the target FPGA. FPGAs as well as logic cells are described in more detail in sub-section 2.3.1.

Implementation

This step implements the netlist, that is generated in the previous step, onto the target FPGA. The implementation can be divided into two main sub-steps, logic optimization and placement & routing of logic cells.

Logic optimization can be considered as a refinement process. The logical design, resembled by the aforementioned netlist, is further simplified prior to placement and routing. The aim is to reduce the area of the design and improve efficiency, before committing physical resources. Furthermore, the optimization can also be tailored to reduce power consumption e.g. Block RAM power optimization.

As for the second sub-step, it revolves around deriving a layout based on the input netlist. Such a layout contains the detailed placement of logic cells and the respective routing of their interconnecting wires, i.e. nets. In other words, it specifies which exact logic cells within the FPGA are used to implement the design, i.e. placement. Thereafter, it specifies their respective connections, i.e. routing.

Based on the netlist and constraints file(s), the corresponding logic cells are placed into appropriate locations within the FPGA. Appropriateness is in-part based on the provided constraint file(s). For example, a timing constraint file is used to identify

which interconnecting wires/signals are time-critical. As such, their respective logic cells are placed in close proximity to one another, thereby achieving shorter time delays, i.e. improving timing efficiency.

Although optional, the second sub-step also includes optimizations. Those are performed twice, once after placement and then again after routing. They can be tailored to reduce power consumption as well as meeting the design's timing constraints.

The aforementioned constraints can be physical, timing or power. For example, physical constraints define the FPGA configuration settings, package pin placement, placement of logic cells or restricting logic cell placement to certain regions of the FPGA. The latter is referred to as floorplanning. As for timing constraints, those define the design's frequencies/periods. Last but not least, power constraints define voltage settings, power and current budgets.

Bitstream

In this step, the aforementioned layouts are converted and consolidated into one output, called Bitstream. This output is also referred to as bit file, configuration file or device image. In short, it contains a specific pattern based on which logic cells and interconnects are configured inside the FPGA. In other words, this pattern specifies which logic cell must be configured and its respective state, i.e. which part of the logic cell is used and how it operates. Furthermore, it also specifies the interconnect structure, i.e. which connections are enabled. Configuring both the interconnect structure and logic cells, implements the functionality required within the FPGA. Sub-section 2.3.1 covers FPGAs and their internal components in more detail.

2.3 Prototype Boards and Development Kits

One of the main advantages of the CarRing IV communication protocol is being implemented entirely in hardware, more specifically in an FPGA. Although FPGAs are the central components, the development and implementation process includes several other hardware components and concepts. In addition to FPGAs, this section

provides a brief description of the main hardware components and concepts that are crucial to both the development process and the understanding of the work presented in this thesis.

Note that FPGAs used in this work are manufactured by Xilinx. In what follows, the used terminology and figures are based on Xilinx as opposed to other manufacturers e.g. Altera. In other words, terms like Configurable Logic Block (CLB) and logic cell are used instead of their analogous terms Logic Array Block (LAB) and logic element.

2.3.1 FPGAs and development boards

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of CLBs connected via programmable interconnects [38].

The term "field-programmable" refers to the ability of configuring the device in the field as opposed to configuration or customization performed during the device manufacturing or fabrication process. In other words, configuration is performed on a manufactured and prefabricated device. This is in-contrast, for example, to gate-array/structured or standard-cell ASICs. Those are "programmed" via one or more tailored masks, i.e. mask programmable [16]. In short, mask programmable refers to ASIC customization that involves constructing multiple stacking pre-designed layers of metal, silicon and polysilicon with customized patterns, i.e. tailored masks. Some layers form transistors, while others form connection wires, i.e. the interconnects.

As for the term "gate array", it refers to the FPGA's internal components and structure. An FPGA consists predominantly of an array of base cells or generic logic cells with a distributed and flexible interconnect structure. More specifically, in a generic FPGA architecture, CLBs consist of multiple slices, which in turn consist of the aforementioned generic logic cells [26]. Each slice has one set of clock, clock-enable and set/reset signals that are common to its logic cells. Sub-section 2.3.3 covers clocks in more detail.

The CLB/slice/logic cell hierarchy is complemented by an equivalent hierarchy in the interconnect [26]. In short, logic cells have faster interconnect than that which

exists between slices, which in turn is faster than CLBs' interconnect. Furthermore, the exact ratio of CLBs to slices to logic cells depends on the particular FPGA family or FPGA product line.

As mentioned previously, "programming" an FPGA corresponds to configuring or customizing the functionality of its internal logic cells and interconnect structure. Since logic cells are the primary components that implement the functionality of a user design, in what follows a brief overview of their configuration method is provided.

In short, configuring a logic cell amounts to configuring its primary sub-component, which is mainly responsible for driving its output signals. Whether be it logic cells in particular or more broadly logic blocks, there are essentially two variations, LUT-based and MUltipleXer (MUX)-based. The former is the predominantly used approach. Comparing and contrasting both approaches is outside the scope of this chapter. Within the context of the work presented in this thesis, the LUT-based approach is considered in the example below.

As the name implies, LUT-based approach corresponds to a look-up table where input signals represent a table index that drives the desired output. For example, consider a logic function within a user design and its associated truth table. The input signals for the LUT sub-component are configured to match the desired output of the aforementioned truth table. Configuring such input signals depends on the type of FPGA used.

In this thesis, Static RAM (SRAM)-based FPGAs are used. The basic premise is that values are loaded into SRAM cells, which are connected to the aforementioned LUT inputs. Since those cells are SRAM-based, the loaded values persist over time until they are manually modified or the FPGA is powered-down. The entire cell comprises a multi-transistor SRAM storage element, whose output drives an additional control transistor that is either OFF(disabled) or ON(enabled) based on the loaded value [26]. As for the loaded values, they are derived from the configuration file described in sub-section 2.2.2.

Other noteworthy types are antifuse-, Electrically Erasable Programmable Read-Only Memory (EEPROM)- and FLASH-based FPGAs. Similar to SRAM-based FPGAs,

the latter two can be repeatedly reconfigured, while the former is One-Time Programmable (OTP). In case of antifuse, configuration amounts to identifying and applying high voltage to physical links associated with configurable paths within the FPGA. In their default state, those links or antifuses correspond to an open circuit which can be closed by applying the aforementioned high voltage.

As for the other two types, their approaches to configuring an FPGA are comparable, with FLASH being the faster and newer technology. Instead of a link, the circuit contains a transistor-based memory cell whose basic structure is comparable to a standard Metal-Oxide Semiconductor (MOS) transistor. In contrast to antifuses, applying high voltage charges an internal gate within the transistor which in turn results in the cell resembling a logic 1. Discharging the aforementioned gate erases the cell.

With the FPGAs' preliminaries covered, in what follows, a brief description of the prototypes and used development boards is provided. While the descriptions below do not encompass all details, they are sufficiently adequate for the purposes of this chapter. In order to avoid repetition, the aforementioned details are covered in the appropriate chapters below.

As mentioned previously, two prototypes were created in the CarRing project. Those differ drastically in their features and more importantly in the amount of FPGA resources available. Briefly describing those prototypes as well as their differences, provides essential preliminary information for the protocol implementation and design decisions described in subsequent chapters.

The first prototype consists of a preexisting product (Xilinx Virtex-5 Mini-Module Plus) and a complementary custom board. The boards are interconnected using two FPGA Mezzanine Card (FMC) connectors. In addition to the FPGA (XC5VFX70T-2FF665), the mini-module also contains dedicated memory modules and interfaces (e.g. Ethernet). As for the custom board, it provides power supply (12V) and contains the main data transmission interfaces used with the Multi-Gigabit Transceivers (MGTs). Moreover, it also contains multiple other interfaces (e.g. CAN, RS232).

In the initial phase of the project, the base node design and development was the

main focus. For that purpose, the Virtex5-prototype has sufficient FPGA resources for implementation and testing. However, as soon as the project progressed further in terms of protocol features and developing the router design, those resources were no longer sufficient. Consequently, a considerable amount of time was dedicated to redesigns that focused on reducing resource consumption rather than the main project goals i.e. the protocol and its feature set. This was the main drive behind creating another prototype.

The second prototype consists of one board created by a third-party manufacturer and further modified to match the project requirements. In contrast to the first prototype, it uses a 5V power supply and includes intermediate interfaces (RaspberryPi, MikroBUS and PMod) instead of directly incorporating specific interfaces (e.g. Ethernet). As such, it offers more flexibility and a wider range of third-party attachments, which in turn contain specific interfaces. In addition to more resources within its FPGA (XC7K160T-2FBG676), more lanes are connected to its main data transmission interfaces, thus resulting in higher bandwidth. In short, a lane is a transceiver sub-component that manages the physical connections to the transmission medium.

Last but not least, additional non-FPGA and FPGA-based kits and boards were used during the CarRing IV project's development. They facilitated the development and realization of one of the project's other goals, which is to transparently transfer data belonging to other networks (e.g. CAN, Ethernet). This allows the project's solution to be seamlessly integrated into existing systems, or serve as a backbone for other networks thereby easing the transition into a pure CarRing-based system.

Non-CarRing-protocol specific designs were developed to control and communicate with other networks' interfaces. Such designs largely reside on top of the protocol's application layer. Some of them were developed using other FPGA-based kits in order to better isolate possible design issues, since the prototypes were still not finalized i.e. a work-in-progress. Both non-FPGA and FPGA-based kits were used for testing and verification. While being noteworthy, a detailed description of those additional kits and boards is outside the scope of this thesis.

2.3.2 IP Cores and ICs

In order to provide the features described in chapter 5, the CarRing protocol must be able to communicate with and control other hardware components, i.e. other chips and interfaces that are external to the FPGA chip within the prototypes.

As briefly mentioned in the previous section, separate designs must be developed and configured into the FPGA. In short, those designs drive specific FPGA I/O pins, which in turn are connected to other chips within the prototypes. Not all such designs must be created from scratch. There exists a similar principle to the Do not Repeat Yourself (DRY) principle from software development, i.e. design re-use. Rather than classes, libraries or frameworks, this involves reusing pre-developed and well-tested hardware components. Whether be it in netlist or HDL form, such components are typically referred to as Intellectual Property (IP) cores and their eventual physical representation being Integrated Circuits (ICs).

Within the context of FPGAs, IP cores can be classified into soft, firm and hard [26]. The latter physically occupies a certain portion or area of an FPGA, i.e. an IC that is heavily optimized and integrated as part of the FPGA chip itself. As such, hard IP cores can be directly included within HDL-level designs. In other words, they do not undergo synthesis and implementation as described in sub-section 2.2.2. An example of such IP core is the Tri-Mode Ethernet MAC used to control the Ethernet interface within the Virtex5-prototype.

Unlike hard IPs, firm cores are not physically integrated into an FPGA. Rather, they are designs that have already underwent synthesis and implementation. Thus, multiple instances can be instantiated and included in a design. As such, the limiting factor is the availability of FPGA resources rather than number of integrated physical components. Firm IP cores are typically represented in netlist form.

As for soft IP cores, they are HDL-level designs that require synthesis and implementation. In the case of SystemC designs, they require an additional step in-which a functionally equivalent RTL Verilog or VHDL design is generated as described in sub-section 2.2.1. All three IP core variations are used within the CarRing project.

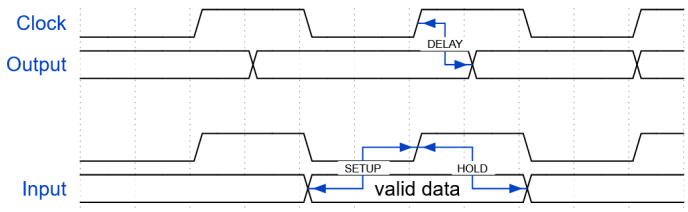


Figure 2-1: Serial Management Interface Timing

Within a design, ICs are typically associated with at least one IP core. They have varying degrees of complexities. In short, lower complexity ones offer less functionality but more flexibility, since the control logic can be created, modified and updated separately within the FPGA. On the other hand, higher complexity ICs offer more functionality and require less FPGA resources, since the control logic typically revolves around interrupts and register manipulation.

Whether its an IP core or IC, controlling and communicating with such components typically consists of two main parts, implementing and handling their respective control interface(s) followed by interrupt management and/or register manipulation. Regardless of the component's complexity, it always has at least one control interface, also referred to as management interface, e.g. Inter-Integrated Circuit (IIC), Serial Peripheral Interface (SPI) or Serial Management Interface (SMI). Continuing the earlier software development analogy, a management interface can be viewed as an Application Programming Interface (API).

The implementation of a management interface must abide by timing diagram(s) described in the component's documentation, called data-sheet. Note that both ICs and IP cores are accompanied by such a documentation. Figure 2-1 shows the timing diagram of an Ethernet's control interface within Virtex5-prototype. Such diagrams are described in the component's electrical specification and depict the behavior of the interface's signals. They typically consist of input signals, which are driven by a custom design, and output signals that are driven by the IC or core in question.

There are varying degrees of difficulties when interacting with an IC or IP core. Depending on the component, management interface implementation must be additionally accompanied by HDL processes that handle registers and interrupts. The

latter can be in form of dedicated interface signals accessible through one or more IC pins, or in the form registers within the IC. Similar to the management interface, register manipulation is also documented in the functional description of an IC's data-sheet.

Furthermore, ICs have additional requirements and conditions. For example, an IC can have an initiation phase that in-part translates into a waiting period, where interaction with its management interface must be delayed until the time interval expires.

2.3.3 Clock Signals and Generators

As described in section 2.1, HDL-level design processes are driven by any change within signal(s) in their respective sensitivity lists. An HDL-level process is typically driven by one special type of signal, called clock signal. It is a periodic digital signal that alternates between two signal levels, a high level which corresponds to a bit value of 1, and a low level which corresponds to a bit value of zero. Within the CarRing project, all design components are driven by clock signals.

Typically, a clock signal is described by its clock period (in seconds), frequency (in Hertz) and duty cycle (in percentage). The latter indicates the percentage of the clock period where the signal is at a high logic level. Within a process, data signals are typically sampled and thereafter used, when the corresponding clock signal transitions from low to high, or high to low levels. These transitions are called rising and falling edge. Collectively, they are referred to as a clock edge.

In order for a design to operate properly, the process's logic operations must be concluded within the clock's period. More specifically, logic operations must abide by the clock signal's setup and hold time constraints. Those constraints refer to the time intervals before and after a clock edge. In short, setup time refers to the time interval, with respect to the clock period, where a stable value of a data signal must be set before a clock edge. Whereas, hold time refers to the time interval where the data signal's value must be held after the clock edge.

Designs have a lower limit on the clock periods that they are able to support.

Such a limit is typically determined by a design's longest data path. In other words, the longest data path, within a design, determines the smallest achievable setup/hold times, thereby determining the lowest clock period supported. Using an unsupported clock period, i.e. lower than the design's limit, results in a timing violation. In this case, the data path must either be modified to match the faster clock signal or a negative slack is encountered. The latter refers to the negative value obtained due to a data path delay being longer than the clock period. On the other hand, a design can be seamlessly switched to a faster clock as long as no negative slack is encountered.

As mentioned in the previous sub-section, the CarRing project incorporates multiple IP cores and ICs into its design. Those use different clock signals and operate with different clock frequencies, which results in a multi-clock domain design. In short, a clock domain refers to a portion of a design, where components are driven by the same clock signal. If all components of a design are driven by the same clock, then such a design has one clock domain.

The CarRing project has a multi-clock domain design. One of the main difficulties of such a design, is passing signals between different domains. In other words, signal values must be correctly set and passed into another clock domain without violating the timing constraints of either domains, i.e. source and destination. This is where Meta-stability is encountered. If setup and hold times are violated, then the transitioning signal can be in a meta-stable state, where its logic value is neither 1 nor 0. In short, this translates to a transistor that can not be reliably set to a voltage value that correspond to logic 1 or 0.

Several solutions can be used to handle single signal transition as well as data transfer between clock domains. Those range from multi-stage (re-)synchronization to independent-clock FIFOs. Although they are used in the CarRing project, covering such solutions is beyond the scope of this thesis.

Note that despite being technically a third signal state, this meta-stable state is separate and apart from high impedance or 'Z' value. The latter is an expected and used state/value, when handling tri-state signals. Such signals are typically used to carry values in both directions between design components, more specifically, using

FPGA pins with I/O buffer support. Tri-state signals are also used in the CarRing project.

With the exception of an internal ring oscillator, a clock signal is typically provided by a clock generator or synthesizer, which is an IC that is external to the FPGA. With respect to the internal oscillator, it outputs a clock that is typically used for FPGA configuration. Such a clock signal typically has a 50% tolerance, i.e. not accurate enough for the CarRing IV project's design. Note that not all FPGAs include an internal clock source.

The project's prototypes include multiple clock generators, which are used both directly and indirectly by the project's design. In the latter case, they are used indirectly by other interfaces within the prototype. For example, the main clock that drives the protocol design is generated by an IP core that uses external clocks provided by a clock generator. This IP core is associated with the prototype's MGTs, i.e. the main clock is a useful by-product of the core and not its main purpose. In subsequent chapters, the main clock is referred to as design-clock, while the MGT clocks are referred to as aurora-clocks.

Furthermore, there are two main types of clocks, single-ended and differential. In short, differential consists of two complementary signals as opposed to one. As for the clock pulse value, i.e. logic 1 or 0, it is derived based on the difference between the two corresponding voltage levels of the respective signals. Moreover, differential clocks have less skewness and jitter than single-ended. Therefore, they are typically used for high speed interfaces. The aforementioned design-clock is a single-ended one, while the aurora-clocks are differential.

2.3.4 Reset Logic

In addition to clocks and clock management, reset logic is another important and crucial aspect of an FPGA design. Mishandling reset logic could lead to hard-to-detect design issues. Furthermore, mismanaged reset logic could negatively impact performance and/or results in a suboptimal usage of FPGA resources.

A design's reset state is typically engaged on system power-up. It could also be

invoked manually via an external source, e.g. a physical button or switch. Additionally, it can be invoked internally by the design itself as an automated response to a specific system event. For example, the detection of multiple consecutive errors, which exceeded a specific error count threshold within a short time interval from a specific interface. In this case, one of the countermeasures implemented into the design could be resetting the interface in an attempt to restore it to an operational state.

Reset logic typically revolves around initializing signals, counters, FSMs, dedicated memory components to a predefined value(s) as well as external ICs to a known state. In the latter, the design initiates the IC's reset operation as described within its datasheet. This usually involves abiding by predefined time intervals and monitoring feedback from the IC in the form of an IC output signal change, and/or a register value change internal to the IC.

For design components entirely contained within the FPGA, reset can be classified as either synchronous or asynchronous. In short, it describes how the incoming reset signal is captured in-relation to the component's clock signal. Each clock domain requires its own dedicate reset logic, in-which the reset signal is accurately captured and synchronized to the domain's clock. Thereafter, the signal's value change is disseminated via intermediate internal signal(s) to other components within the domain.

With respect to reset signals originating from physical buttons or switches, reset logic must also account for the stability of the input. This is separate and apart from possible reset signal timing violations or meta-stability issues. In short, an additional dedicated component, typically called debouncer, is implemented to handle possible input signal frequent fluctuations over a short time interval. It acts as a form of a barrier, where frequent consecutive changes in reset signal value are captured and filtered until the signal settles on a specific value, i.e. either logic 1 or 0.

Within the CarRing IV project, the design's reset is not only connected to the input pin of a physical button, but also affected by the reset logic of the MGTs' IP cores as well as the status signals of the MGTs themselves. Depending on the clock domain and design component in-question, asynchronous, synchronous and hybrid

reset solutions are used within the project. In the latter, the input reset signal is de-asserted synchronously to the corresponding domain's clock and asserted asynchronously. Assertion refers to the active state of the signal, while de-assertion refers to its inactive state. As for the logic value associated with a signal state, it depends on whether the signal is active high or active low. Within the context of a reset signal, reset is indicated by a logic 1 in the former and logic 0 in the latter.

Chapter 3

Industrial networks and real-time communication

This chapter gives an overview of the current real-time networks used in automotive, avionics and factory automation. More specifically, it focuses on the used real-time approaches and communication paradigms. In addition to event- and time-triggered communication, the first section also describes the approaches used to achieve and maintain real-time behavior. Furthermore, it describes the scope of the chapter, i.e. the selection criteria based on which the aforementioned networks are included. Thereafter, each network is covered in a dedicated section. Finally, a summary is provided.

3.1 Introduction

Prior to delving into the real-time approaches, isochronous real-time must be defined. As described in section 1.1, real-time systems can be classified into soft, firm and hard real-time. Within the context of real-time communication, network traffic can also be classified into real-time and non-real-time as well as isochronous real-time. The latter can be defined as delivering user data at a specific time instant, i.e. more strict than having a hard upper limit on latency, which is the case for hard real-time. Within the context of Profinet IO [33], it corresponds to cycle-based communication in-which

individual data frames are transmitted at time instances that are equidistant with respect to consecutive communication cycles.

3.1.1 Real-time approaches

There are several approaches to achieving and maintaining real-time behavior, i.e. adhering to timing constraints. Those approaches manage and control access to the underlying transmission medium.

Priority-based arbitration

As the name implies, timing guarantees are provided by prioritizing messages, i.e. data transmissions. Such priorities are assigned based on either the deadline association with the message, time spent by the message waiting in the transmission queue or time-criticality of the message, e.g. isochronous traffic having the highest priority.

Alternatively, priorities can also be assigned by the user application or based on the properties of the frame/packet carrying the message. In the latter case, priorities are assigned dynamically at the time of arrival of the frame/packet. Within the context of connection-oriented protocols, the assignment can also be made statically for all packets/frames in a connection.

In addition to transmission order, those priorities can also affect which packets/frames are discarded in case of a full transmission queue.

Rate-constrained or Bandwidth reservation

This method revolves around reserving available network resources to incoming traffic. Reservations can be made either statically or dynamically.

In the static case, network resources are split into multiple classes each with its own Quality of Service (QoS). As such, timing guarantees are provided based on the class of the message. Another variation is to split the underlying communication channel/link into multiple virtual ones. Thus, isochronous real-time, real-time and non-real-time traffic are isolated from one another. And each virtual channel/link has

its own level of QoS.

As for the dynamic case, the sender attempts to reserve resources prior to transmitting messages. This process of resource reservation is called Admission Control. In short, the sender first checks for the availability of the required network resources along the message path, i.e. within routers, switches, etc. If the required resources are available, then the resources are allocated. Otherwise, the sender might attempt to renegotiate with a lower QoS constraints. This dynamic reservation is typically applied to a connection i.e. to a group of messages.

Scheduler-based

Timing guarantees are provided based on scheduling system in which usage of network resources remains below 100% i.e. no over-commitment. Such schedules are typically managed/enforced by a central node within the network, called controller node. Additionally, the controller node is also responsible for network-wide time synchronization. In short, local clocks within network nodes are synced based on a reference node, which is typically the controller node itself. The synchronization can be time-stamp based or relative timing with respect to the reference node/clock.

Alternatively, the schedule can be created prior to deployment and manually loaded into network nodes before run-time. In such a case, the controller node is still responsible for time synchronization.

Analogous to real-time systems, schedulers can be preemptive or non-preemptive. In the former, an active transmission might be preempted in which the corresponding packet is replaced by a newly arrived one prior to transmission start. The replacement is based on priorities, i.e. similar to priority-based arbitration. Note that preemption can not take place if transmission already started, i.e. while a packet is being transmitted. Otherwise, it would result in a failed transmission. This is in stark contrast to real-time systems in which schedulers are able to interrupt ongoing tasks during their execution and resume them thereafter.

In the previous approach, QoS constraints are grouped together into classes. Whether they are statically or dynamically assigned, this results in a less fine control over un-

derlying constraints. Whereas in this approach, schedules can be created where the main focus is on a specific QoS constraint e.g. delay, jitter. Thus allowing a more granular control with respect to resource allocation.

Time-slicing or cycle-based

In this method, data transmission is organized into periodically repeated send cycles. Each cycle is split into multiple time-slots. As such, timing guarantees are provided based on bounded access, i.e. time-slots.

Time-slots can be based on the number of nodes within a network. In such a case, each node has its own time-slot within a send cycle. This allocation can be static or dynamic. In the former, the time-slot is dedicated to its node regardless whether it has user data to transmit. As for the latter, the time-slot can be used by another node if its owner is idle.

Moreover, the size of a time-slot corresponds to a certain level of QoS. As such, time-slot sizes can be set based on the corresponding node's requirements i.e. the requirements of the user application(s) associated with the node. In other words, nodes are assigned adequately sized time-slots. Thus, time-slots do not necessarily need to be equally sized i.e. their sizes can be varied.

Regardless, network-wide time synchronization is required in order for nodes to accurately start data transmission within their respective time-slots. Similar to the previous approach, the synchronization can be time-stamp based or relative timing based on a reference node.

Last but not least, the approaches described above are not mutually exclusive. For example, the cycle-based approach can be combined with the priority-based arbitration. Within a time-slot, the corresponding node transmit packets based on their respective priorities. Another example is combining scheduler and cycle-based approaches. In such a scenario, the send cycle is split based on time-criticality of the message. And nodes are allowed to transmit their messages based on the schedule.

3.1.2 Event- and time-triggered communication

In short, event-triggered communication corresponds to data transmission being driven by or in response to the occurrence of an event. As the name implies, time-triggered communication is time-based and requires network-wide time synchronization within the context of real-time communication.

The real-time approach used in a network also affects the communication paradigms supported i.e. event-, time-triggered or both. With respect to the approaches described above, priority-based arbitration and rate-constrained are more compatible with event-triggered communication. On the other hand, scheduler- and time-slicing approaches are more compatible with time-triggered communication.

As mentioned in the previous sub-section, those approaches are not mutually exclusive. Thus, support for event-triggered communication can be introduced into a network where communication is time-triggered. For example, adding a "dynamic" slot in a network that uses time-slicing approach, where nodes can transmit based on the availability of the underlying communication channel/link, thereby adding support for event-triggered communication.

3.1.3 Selection criteria

Within the context of computer networks, there exists a wide variety of solutions, i.e. standards and protocols. In order to provide references that are relevant to the work presented in this thesis, a selection criteria must be established.

- The solution must be current and hard real-time capable, or soft real-time capable that is used in time-critical applications e.g. x-by-wire. This is due to the impact of providing real-time guarantees on the achievable performance metrics when compared to the capability of the underlying transceivers and transmission medium. Thus, solutions aimed at best-effort networks are not an adequate comparison. Note that the term "current" refers to the solution still being accepted/valid and not whether it is wide-spread. In other words, it is not based on market-share or adoption level in the industry or academia.

- The solution must include and be able to transmit payloads i.e. can carry user data, not signals.

- The solution must be complete i.e. include a user interface as well as a physical layer interface. For example, single-layered solutions are not adequate for comparison. Moreover, assembling protocol suites/stacks based on existing partial solutions is not attempted in this chapter. While the network performance can be inferred in such a scenario, the assembled solution is not tested in its entirety. However, this does not exclude preexisting and tested solutions that are based on multiple partial solutions i.e. protocols suites.

- If the solution has multiple versions/variations/classes/profiles, then the one with most deterministic behavior and time guarantees is selected.

- Wireless networks and their respective solutions are excluded. This is due to the nature of the underlying transmission medium and the challenges that arise from it with respect to timing guarantees. An open medium is subject to interference and is more difficult to be physically contained. Therefore, it can be argued that wired networks have an advantage over wireless networks with respect to providing timing guarantees. As such, wireless networks are considered as a separate category that is outside the scope of this chapter.

In subsequent sections, the covered standards/protocols are organized based on their primary networks i.e. the main networks in which they are used. The specifics and internal operation of the networks as well as their respective standards/protocols are not covered. In other words, no descriptions pertaining to packet/frame formats or user data processing within layers. For such information, there is already an existing wide body of work for it. In this chapter, the focus is on the approach with which real-time behavior is achieved and maintained as well as the communication paradigm.

3.1.4 Ethernet, determinism, PLCA and TSN

Ethernet-based solutions should not be automatically considered as being not deterministic. Within the context of Standard Ethernet, worst case packet/frame latency can not be bounded due to the way collisions are handled. In case of a collision, nodes pause for a random period of time before attempting to retransmit. While the probability of consecutive collisions is reduced, it is not eliminated. As such, real-time capable Ethernet-based solutions either use a modified Ethernet, or design protocols that ensure collision-free data transmissions. In the latter, a different communication method is used, i.e. not Carrier-Sense Multiple Access with Collision Detection (CSMA/CD).

A noteworthy and relatively recent example of modified Ethernet is PHY-Level Collision Avoidance (PLCA), which is defined in clause 148 of [10]. It is a reconciliation sublayer that addresses the non-deterministic behavior of the CSMA/CD method. In other words, PLCA improves CSMA/CD rather than replacing it. Using this sublayer, physical collisions are avoided, while providing an upper limit on latency. It revolves around dynamically managing sending opportunities such that nodes transmit one at a time, thereby avoiding collisions. PLCA uses the cycle-based approach.

Within the network, nodes are assigned unique identifiers. The node with identifier value of 0 is referred to as coordinator. It is responsible for managing communication cycles, i.e. signaling to other nodes the start of a new cycle. In a round-robin manner, each node gets one sending opportunity per cycle. If a node does not transmit within a certain time limit, then the sending opportunity is yielded, thus proceeding to the next node. Note that transmitted packets can vary in size i.e. no fixed length.

On the other hand, several protocols have been developed which aim at achieving/improving the deterministic behavior of Ethernet. They have been introduced as proprietary solutions and standards. Of particular interest are the standards introduced by Time-Sensitive Networking (TSN), an IEEE task group. Due to the broad focus of the group, in what follows, only a subset of those standards are covered. Within TSN terminology, they are referred to as traffic shapers. In short, they are a

set of mechanisms that control the flow of egress traffic in order to provide timing guarantees.

- **IEEE 802.1Qbv - Enhancements for Scheduled Traffic**. This standard defines the Time-Aware Shaper (TAS). Within TSN traffic shapers, it provides the most strict timing guarantees i.e. used for isochronous transmissions. As for the underlying real-time approach, it is primarily cycle-based. Data transmissions are organized into periodically repeating fixed size communication cycles. Time critical data are assigned dedicated time-slots, thereby guaranteeing uninterrupted transmission. As for non-critical data, priority-based arbitration is used. Isolating the traffic is made possible using a time-aware scheduler. It operates at the port level, wherein traffic is organized into time-gated queues.

- **IEEE 802.1Qch - Cyclic Queuing and Forwarding**. Unlike the previous shaper, the Peristaltic Shaper (PS) of this standard operates within switches. The main objective is to bound the frame residence time within the switch. As such, latency bounds depend in-part on the number of hops traversed during transmission. Similar to TAS, the underlying approach is cycle-based. However, incoming traffic is processed based on its priority and time of arrival. As for the communication cycles, they correspond to an alternating and equally sized time intervals, labeled "even" and "odd". Within the switch, incoming frames are labeled based on the time interval in which they have arrived. And they are transmitted in the next time interval, i.e. "even" frames are transmitted in the subsequent "odd" time interval and vice versa.

- **IEEE 802.1Qcr - Asynchronous Traffic Shaping**. The approach used by Asynchronous Traffic Shaper (ATS) closely resemble priority-based arbitration. Similar to the previous standard, the deterministic behavior is primarily maintained by switches, whereas network nodes are required to adhere to specified rate limits. ATS uses Urgency-Based Scheduler (UBS), which processes incoming traffic using a per-flow shaped queuing scheme followed by a per-class queuing scheme. In the former, frames are classified into per-flow shaped queues

based on fixed-priority assigned by the upstream source i.e. the 3-bits Priority Code Point (PCP) of the Virtual LAN (VLAN) tag. Thereafter, the latter uses Internal Priority Values (IPVs) to classify/merge frames into per-class shared queues. In turn, UBS assigns eligibility times to frames, which are then used to indicate when the next frame can be transmitted. UBS uses an interleaved scheduling algorithm, called Token Bucket Emulation (TBE), in order to achieve asynchronous traffic shaping with low delay. It is derived from the token-based leaky bucket algorithm. In short, the time value corresponds to the time needed to accumulate enough tokens to transmit a frame. The number of tokens required depends on the frame's length. At the egress port, frames are selected for transmission using strict priority queuing, which prioritizes transmission of frames in higher priority queues over lower ones.

The first two shapers require network-wide time synchronization. Note that they only specify that synchronization is required. As such, they can be combined with another standard introduced by TSN, IEEE 802.1AS-rev Timing and Synchronization. On the other hand, ATS does not require time synchronization.

Last but not least, TSN traffic shapers also includes Credit-Based Shaper (CBS) defined in IEEE 802.1Qav. However, it's capable of soft real-time transmission and is primarily used for audio/video broadcasting. In other words, it must be combined with another mechanism, e.g. Strict Priority [44], in order to provide timing guarantees that are sufficient for industrial networks.

3.2 Automotive

Vehicular communication or networks can be divided into three main categories, Vehicle to Infrastructure (V2I), inter-vehicular and intra-vehicular. The former two can also be grouped together as Vehicle to Everything (V2X). And they are based on Wireless LAN (WLAN) and cellular i.e. wireless networks. Therefore, in what follows, the solutions covered are specific to intra-vehicular communication, i.e. wired networks.

Such a network employs a wide variety of standards and protocols. Each one is aimed at a certain level of timing and performance requirements. Therefore, the network is split into multiple smaller networks based on the requirements and standards used. In other words, each smaller network employs one standard.

These smaller networks are interconnected by a specialized component, called gateway. Using a gateway, nodes belonging to different smaller networks are able to communicate. Intra-vehicular networks either use one central gateway or multiple smaller gateways, referred to as domain gateways. While being an essential component of the network, gateways are outside the scope of this chapter. Although they involve a routing-like functionality, they are considered as a workaround solution addressing the multiple incompatible standards used.

3.2.1 CAN/-FD, MilCAN A

Within the context of medium access control protocols, CAN [4] uses Carrier-Sense Multiple Access with Collision Resolution (CSMA/CR), which is non-deterministic. Nevertheless, determinism can still be provided for high priority messages. The priorities are assigned to the messages as opposed to the nodes transmitting them. Each CAN message is equipped with a unique identifier which is the first value sent upon transmitting a message. It is used as the priority value as well as the means by which other nodes distinguish and receive the message. The latter is a filter-based approach, i.e. no explicit destination address.

The underlying network topology is a shared bus in which all network nodes are able to monitor and receive any message being transmitted. In other words, CAN uses broadcast/multicast transmission. As such, a collision can occur if multiple nodes transmitting at the same time. To provide timing guarantees, the approach used is priority-based arbitration, which takes place in case of a collision. Transmission of lower priority message is stopped in favor of the highest priority message. Note that the conflict resolution method is not destructive i.e. lower priority messages are still transmitted by their respective nodes thereafter. However, due to the arbitration method used, low priority messages may starve.

With respect to communication paradigm, CAN is event-triggered in which messages can be transmitted periodically or sporadically based on an external trigger. And network-wide time synchronization is not a requirement for CAN's core functionality. As for performance, it can reach up to 5 Mbps with CAN Flexible Datarate (CAN-FD) [3].

Within the context of vehicular networks, there are other variations of CAN that employ different approaches to provide timing guarantees e.g. MilCAN A [30]. In this case, nodes transmit messages based on a predefined schedule, which bounds the latency of each message. In turn, the schedule is based in-part on using sync slots, each with its own identifier. Nodes periodically receive a sync frame/message, which contains the current active sync slot number. This number is incremented with each sync frame transmitted, 0 to 1023 and repeats.

3.2.2 FlexRay

To provide timing guarantees, FlexRay [6] incorporates multiple approaches, wherein time-slicing serves as the primary approach. Data transmission is organized into periodical communication cycles. Each cycle consists of static, network idle time, dynamic and symbol window segments. The former two segments are mandatory, while the latter two are optional.

Data transmission takes place within the static and dynamic segments of a communication cycle. Only one network node transmits at a time, while others wait until it finishes. Scheduling is used within the aforementioned two segments. Statically scheduled frames are transmitted in the static segment (bounded latency), whereas the dynamic segment employs the mini-slotting scheme to transmit dynamically scheduled frames. This scheme is also known as Flexible Time Division Multiple Access (FTDMA).

Within the static segment, frames are transmitted using equally sized static slots. A node can have multiple slots within a communication cycle. As for the dynamic segment, a dynamic slot consists of one or more mini-slots. A node is assigned multiple mini-slots based on the length of the frame being transmitted. If the node has no

frame to transmit, then it is assigned one mini-slot and the rest can be used by other nodes. Therefore, dynamic slot length can vary. This allows for dynamic bandwidth allocation.

With respect to communication paradigms, FlexRay supports both event- and time-triggered. The standard implements distributed clock synchronization and can be configured to operate in event- and time-triggered modes. In the latter, the communication cycle consists of either only static slots or a mix of both static and dynamic slots. As for event-trigger mode, one static slot is used while the rest of the slots are dynamic. In this case, priority-based arbitration is used within the dynamic slots. The priorities are based on time-offset values i.e. higher priority messages are allocated mini-slots that are closer to the beginning of the dynamic segment. If the number of frames exceeds the segment length, then the lower priority messages are transmitted in the subsequent communication cycles.

FlexRay supports both star and bus network topologies. As for performance, it can reach up to 10 Mbps.

3.2.3 MOST150

In Media Oriented Systems Transport (MOST) [8], timing guarantees are primarily provided using the bandwidth reservation approach. There are multiple versions of MOST, namely MOST25, MOST50 and MOST150. The latter added support for isochronous transmission.

In essence, the underlying communication channel is split into control, packet data and streaming data channels. The latter two are used to transmit user data, while the former is used for network administration, i.e. control commands and status messages. Since it is shared by all network nodes, no guarantees can be provided within the packet data channel. On the other hand, streaming data channel supports real-time transmission of synchronous and isochronous data. In both cases, a connection is established before data transmission starts, thereby reserving network resources. Thereafter, Time Division Multiplexing (TDM) is used wherein data is transmitted cyclically based on a specified time pattern.

The streaming data channel is mainly used for audio/video data. The isochronous data transfer eliminates the need for user applications to synchronize their streams to MOST , i.e. it's time base. Within the context of isochronous data, three mechanisms are supported, Audio/Video packetized, DiscreteFrame and QoS IP streaming. The former two are aimed at audio and video data, while the latter is used for QoS-based packet transmission. In a QoS IP channel, a user application has exclusive access to the reserved bandwidth.

MOST also implements time synchronization. One dedicated node continuously transmits frames that synchronize other nodes within the network. Thus, nodes are continuously resynchronizing using the preamble at the beginning of the frame transfer. Despite being a shared bus, MOST network is typically uses a (logical) ring topology. With respect to performance, it can reach up to 150 Mbps.

3.2.4 LIN

Using the scheduler-based approach, Local Interconnect Network (LIN) [7] is able to provide timing guarantees. Prior to deployment, a schedule is generated that contains all transmissions. During run-time, a dedicated network node is responsible for, i.e. manages, the schedule. This node is referred to as the LIN master node. Transmissions can only be initiated by the master node, wherein the polling method is used. In short, a LIN frame consists two parts, header and response. They are transmitted separately. The header is always sent by the master node, while the response is either sent by another node or the master node itself.

LIN also implements time synchronization. Using the header part of the frame, other nodes are synchronized to the master node's timing. As such, LIN uses the time-triggered communication paradigm. With respect to network topology, LIN is a shared bus. As for performance, it can reach up to 20 kbps.

3.3 Avionics

In avionics, nodes are primarily interconnected via switches within a meshed network. This is in contrast to automotive networks, which are primarily shared buses interconnected by either a central gateway or multiple domain gateways. Unlike automotive, Ethernet is adopted and used for safety critical applications, i.e. fly-by-wire.

3.3.1 TTCAN, CANaerospace

In avionic networks, CAN is used in a relatively wide range of systems ranging from flight control to passenger comfort. Since CAN is already covered in sub-section 3.2.1, in what follows, other used variations of CAN are described, namely Time Triggered CAN (TTCAN) [18] and CANaerospace [35].

TTCAN adds support for time-triggered communication paradigm to standard CAN. It primarily uses scheduler-based approach to provide timing guarantees. Within the network, a dedicated node, referred to as time master, is responsible for the schedule as well as local/global time synchronization.

In essence, the schedule is organized as a matrix where each row corresponds to a Basic Cycle (BC), i.e. a communication cycle. As for columns, they are organized into time windows. Although time window sizes can vary within a row, the size must be the same across rows, i.e. fixed size columns. Thus, all BCs consists of similarly sized time windows, wherein the window type can vary.

There are three types of windows, exclusive, arbitration and free. The former two are used for data transmission, while the latter is reserved for possible TTCAN expansions. The exclusive window is reserved for a predefined message. Whereas message transmission within the arbitration window is based on standard CAN. Thus, both communication paradigms are supported by TTCAN.

Data transmission is initiated by the time master. Each BC starts with a special message sent by the time master, called Reference message. This message is used for time synchronization as well as to indicate which BC is currently active. Using the cycle count, nodes transmit their messages based on the schedule.

As alluded to above, TTCAN implements two levels time synchronization. The first level is based on a 16-bits counter that is restarted with each reference message, i.e. with each BC. Whereas the second level is based on a 19-bits counter that is synced to the master node's time.

Similar to TTCAN, CANaerospace also uses a scheduler-based approach to provide timing guarantees. In contrast with TTCAN, it uses a relatively simplified approach that does not require a dedicated master node. As such, it is required that each network node abide by its transmission schedule.

Data transmission is organized as fixed size time slots, called minor time frames. Thus, the number of transmitted messages is limited within the boundaries of a time slot. However, the number of messages transmitted by each node may vary. As for the time slot size, it is calculated along with the schedule prior to deployment. As such, CANaerospace uses the time-triggered communication paradigm.

3.3.2 AFDX, uAFDX

Despite being Ethernet-based, Avionics Full-Duplex Switched Ethernet (AFDX) [32] achieves determinism using the rate-constrained approach. The underlying communication channel is divided into multiple logical links, called Virtual Links (VLs). Each VL has its own identifier, Bandwidth Allocation Gap (BAG), max/min frame size and a bandwidth limit.

The identifier is incorporated into the MAC address. More specifically, it replaces the last 16-bits i.e. used in delivery as part of the destination address. BAG is used to manage possible congestion within the network. It defines the min delay between sending two consecutive frames. By defining max/min frame sizes, fragmentation is no longer needed, thereby eliminating a possible source of non-determinism. Combining the bandwidth limit with the aforementioned VL properties, bandwidth can be guaranteed for each VL.

While also used by network nodes, those VL properties are maintained and enforced by specialized AFDX switches. In other words, the deterministic behavior is primarily maintained by the AFDX switches as opposed to nodes. AFDX uses the

event-triggered communication paradigm. With respect to network topology, it uses the star topology. As for performance, the underlying bandwidth can reach up to 1 Gbps.

A noteworthy extension of AFDX, called uAFDX, can also be used in automotive networks. It uses the same switching/VL concept as AFDX. The main distinction is in the functionality of the specialized switches. The uAFDX switch operates as a hub for down-links and as an AFDX switch on up-links. While using a star topology, an uAFDX network is functionally comparable to a bus.

3.3.3 TTP/C

In Time Triggered Protocol class C (TTP/C) [11], timing guarantees are primarily provided using the time-slicing approach. Data transmission is organized into periodical communication cycles, called cluster cycles. In turn, a cluster cycle is further divided into equally sized Time Division Multiple Access (TDMA) rounds. Within a TDMA round, each network node is assigned a time slot in which it can transmit messages. The size of such a time slot is fixed across rounds. However, it can differ from one node to another. As such, a node can transmit different messages in different rounds within its time slot.

In order to identify which message must be transmitted at which round/time-slot, nodes use a static schedule that is generated and locally configured prior deployment. TTP/C implements distributed clock synchronization. And it uses the time-triggered communication paradigm. With respect to network topology, it can be configured as bus or star. As for performance, it can reach up to 25 Mbps.

3.3.4 MIL-STD-1553C

Using the scheduler-based approach, timing guarantees are provided by MIL-STD-1553C [29]. Data transmission can only be initiated by a dedicated node within the network, called bus controller. Using a message-scheduling scheme, it enables other nodes to transmit data. Based on the scheme, the bus controller sends a command,

which in turn triggers the data transmission. Thereafter, the node sends back a response indicating the status of its data transmission. Both the data transmission and the following response have a time limit. As for the commands, they can be sent asynchronously or periodically based on the bus controller's local time, i.e. time synchronization is not a necessity for the core functionality.

MIL-STD-1553C uses the event-triggered communication paradigm. With respect to network topology, it can be configured as bus or star. The latter can reach up to 10 Mbps with respect to performance. Furthermore, a fiber optic version of the standard, called MIL-STD-1773, can reach up to 20 Mbps.

3.3.5 IEEE 1394B

IEEE 1394B [5] has two data transfer modes, asynchronous and isochronous. The latter is used for real-time data transmission. Within this mode, timing guarantees are primarily provided using the bandwidth reservation approach. Data transmission is organized into periodical fixed size (125 us) communication cycles. Within each cycle, bandwidth is reserved as channels i.e. portions of the 125 us time interval. Even when the isochronous mode is engaged, only up to 80% of the total bus bandwidth can be used for isochronous data transfer and the remaining 20% is used for asynchronous mode. The latter corresponds to best-effort delivery.

During the initialization phase of the network, a node is elected and designated as the root node. In addition to arbitration between nodes, it is responsible for managing the communication cycles. As such, it is also referred to as the cycle master. In order to indicate the start of a cycle, it broadcasts a cycle start telegram which also syncs network nodes. Thereafter, if a node needs to transmit data, then it sends a request back to the root node. In case of multiple simultaneous requests, priority-based arbitration is used. Priorities correspond to the proximity of the respective node to the root node, i.e. the closer, the higher.

With respect to communication paradigm, IEEE 1394B is event-triggered. Despite being a bus, it uses tree topology. As for performance, it can reach up to 3.2 Gbps.

3.3.6 TTEthernet

To overcome the non-determinism of standard Ethernet, Time Triggered Ethernet (TTEthernet) [12] manages data transmissions in such a way that prevents collisions, i.e. no arbitration is performed. It supports Time-Triggered (TT), Rate-Constrained (RC) and Best-Effort (BE) traffic. The former two provide timing guarantees using the scheduler-based and rate-constrained approaches, respectively.

In case of TT mode, data transmission is carried out based on a predefined schedule. Prior to deployment, the schedule is generated and configured locally within each network node. During run-time, conflict/collision free transmission is guaranteed by the pre-calculated schedule, i.e. no overlapping time-slots. In this mode, TTEthernet implements a fault-tolerant distributed clock synchronization.

As for the RC mode, data transmission is organized as RC data-flows, i.e. sequence of messages. Similar to AFDX, networks nodes transmit while maintaining a minimum Inter-Frame Gap (IFG) between consecutive RC data-flows. While nodes must abide by minimum IFG, the deterministic behavior is ensured by network switches, wherein transmissions that violate the minimum gap are dropped. No time synchronization is required in this mode. As such, buffer sizes are allocated based on peak-load scenarios in order to prevent possible loss.

Due to the TT and RC traffic classes, TTEthernet supports both time- and event-triggered communication paradigms. With respect to network topology, it can be configured as a star or ring. As for performance, it can reach up to 1 Gbps.

3.4 Automation

There is a wide variety of standards and protocols used in this domain. They can be divided based on their target networks, management level i.e. business and supervisory networks, as well as automation- and field-level i.e. fieldbus networks. In this chapter, the main focus is on standards/protocols used within the latter due to its real-time requirements. Such standards/protocols are used in process and control, where network nodes are typically machines, sensors, actuators and control devices.

Similar to avionics, Ethernet-based solutions are also more broadly adopted and used in time-critical applications. Within those solutions, there is a wide variety of proprietary protocols. This is in-part due to the necessity of providing backward-compatibility to previous already deployed standards. In other words, Ethernet is used for lower layers, while higher layers maintain compatibility with existing/previous non-Ethernet version of the standard. When compared to automotive, gateways are also used to interconnect machines using different standards.

3.4.1 EtherCAT

The approach used by Ethernet for Control Automation Technology (EtherCAT) [19], to provide timing guarantees, closely resemble a scheduler-based approach. Data is transmitted at specific time instants or periodically, wherein time cycles with varied lengths can be used. For example, in the latter, shorter time cycles are used for commands, e.g. refresh local data with network nodes, while longer time cycles are used to retrieve new data.

In EtherCAT, network nodes form a logical ring. Data transmission can only be initiated by a dedicated node, referred to as the master node. It uses the summation frame method, wherein an Ethernet frame, called EtherCAT frame, is used to exchange data between all nodes. Multiple datagrams can be encapsulated within the frame. Each node retrieves/inserts data (if targeted by master) into the frame, and forwards it to the next node within the network. The last node sends the processed frame back to the master node.

Note that there is another variant of this transmission method that uses UDP/IP. This has a larger overhead and it is used for less time-critical applications.

EtherCAT is primarily time-triggered and implements a distributed clock synchronization. While it is logically a ring, it can be physically configured as star, line, tree or daisy-chain. As for performance, it can reach up to 1 Gbps.

3.4.2 PROFINET IO CC-C/CC-D

The PROcess FIeld NET (PROFINET) [33] standard is divided into multiple Conformance Classs (CCs), A through D. Starting with CC-A as the "basic" class, each class supports a certain QoS and set of basic functionality. Since isochronous real-time support is established in CC-C, this class is covered in this section. As for CC-D, it is a TSN compliant version of CC-C, i.e. provides the same functionality.

Timing guarantees are primarily provided using the cycle-based approach. Data transmission is organized as send cycles, which in turn is divided into multiple time intervals or phases. Each phase is aimed at a specific traffic class. Starting with isochronous real-time phase followed by real-time and non-real-time phases. Within the first phase, a scheduler-based approach is used. Data transmission is carried out based on a static schedule that is generated prior to deployment and configured locally within each network node. Furthermore, special PROFINET switches are required to maintain a low consistent jitter. As for the second phase, priority-based arbitration is used, i.e. real-time traffic is prioritized over non-real-time using the VLAN header field.

With respect to communication paradigms, PROFINET IO CC-C is time-triggered. It implements network-wide time synchronization, which also includes the aforementioned special switches. Regarding network topologies, it supports star, line, tree and bus. As for performance, it can reach up to 1 Gbps.

3.4.3 SERCOS III

In SErial Real-time COmmunication System (SERCOS) III [37], timing guarantees are provided using the cycle-based approach. Data transmission is organized into communication cycles consisting of a real-time and non-real-time interval/channel. Within the real-time channel, transmissions are initiated by a dedicated node within the network, referred to as master node. Within a SERCOS III network, nodes form a logical ring.

In the real-time channel, the summation frame method is used to exchange data

between nodes. It is carried out over two phases. First, the master node transmits a Master Data Telegram (MDT), wherein network nodes have read-only access. Thereafter, the master node transmits an Acknowledge Telegram (AT) in which nodes insert data as a response to the former transmission. Up to four MDTs and ATs each can be transmitted within one communication cycle. Those are used in case that the amount of data exceeds the limits of one MDT or AT.

As for the non-real-time channel, it corresponds to the time left within a communication cycle, i.e. after real-time transmissions have concluded. As such, nodes transmit non-time critical data, i.e. status and diagnostic messages.

SERCOS III implements network-wide time synchronization. As for communication paradigms, it is time-triggered. A SERCOS III network can be configured in a line or ring topology. With respect to performance, it can reach up to 100 Mbps.

3.4.4 Ethernet POWERLINK

Timing guarantees are provided by Ethernet POWERLINK [17] using the cycle-based approach. Data transmission is organized into communication cycles which consist of isochronous, asynchronous and idle phases. The latter is optional and corresponds to the leftover time between the end of the second phase, i.e. asynchronous phase and the start of the next communication cycle. As the name implies, no data transmissions are carried out during the idle phase.

Within the network, a dedicated node, referred to as managing node, is responsible for those communication cycles. Furthermore, data transmissions are controlled by the managing node using the polling method. To indicate a new cycle, the managing node multicasts a Start-Of-Cycle (SoC) frame to all network nodes. During the isochronous phase, a node is able to transmit only if it receives a Poll-Request (Preq) frame from the managing node. The Preq frame is sent as a unicast transmission. Thereafter, the targeted node replies with a Poll-Response (Pres) frame. Such a frame is sent as a multicast, thereby replacing target-specific transmission with a producer/consumer model.

Similar to the first phase, the managing node multicasts a Start-Of-Asynchronous

(SoA) frame to all nodes. However, unlike the isochronous phase, only one data transmission is carried out by either the managing node itself, or one of the other nodes within the network. Nodes can include send requests within the aforementioned Pres frame, i.e. during the first phase. In case of multiple requests, priority-based arbitration is used, while maintaining that no send request is indefinitely delayed.

Ethernet POWERLINK implements distributed clock synchronization, wherein nodes are synced based on the managing node's timing. With respect to communication paradigms, Ethernet POWERLINK is time-triggered. And it can be configured in star or line topologies. As for performance, it can reach up to 1 Gbps.

3.4.5 ControlNet

ControlNet [31] provides timing guarantees by using the time-slicing approach. It uses Concurrent Time Domain Multiple Access (CTDMA), wherein network nodes transmit data within a periodically repeating time interval, referred to as Network Update Interval (NUI). This interval is divided into scheduled, unscheduled and maintenance parts.

As the name implies, data transmission within the first part takes place at specific time instants using a predefined schedule. A so-called keeper node is responsible for maintaining and distributing the schedule as well as other network information. During the scheduled part of NUI, scheduled nodes can transmit once per NUI.

On the other hand, both scheduled and unscheduled nodes can transmit within the second part of NUI. Starting with the smallest node address (0), nodes transmit data on a sequentially rotating basis. Within a rotation, one data transmission is allowed per node. This repeats until the time allocated for the unscheduled part is consumed. Last but not least, network nodes are synchronized within the maintenance part.

In a ControlNet network, data is exchanged using the producer/consumer model, wherein transmissions can be carried out as unicast or multicast. Within the context of communication paradigms, ControlNet is time-triggered. And it can be configured in line, tree or star topologies. As for performance, it can reach up to 5 Mbps.

3.4.6 EPA

Ethernet for Plant Automation (EPA) [9] uses a modified data-link layer to achieve deterministic behavior. This translates to a Communication Scheduling Management Entity (ECSME) situated between the network layer and the MAC sub-layer.

Timing guarantees are provided using the time-slicing approach. Data transmissions are organized into communication cycles, referred to as macro-cycles. There are two phases within a macro-cycle, periodic and non-periodic message transfer phases. The sending time of such messages is controlled by the aforementioned ECSME.

Within the first phase, message are transferred at specific time instants based on a pre-configured schedule. Furthermore, the last part of the periodic message transmission includes a non-periodic message announcement. As such, nodes announce whether they need to transmit in the second phase of the macro-cycle. In addition to time availability, priority-based arbitration is used as the main approach within the non-periodic message transfer phase.

EPA uses the time-triggered communication paradigm and implements time synchronization. An EPA network can be configured in line, star, ring or daisy-chain topologies. As for performance, it can reach up to 100 Mbps.

3.5 Summary

In this chapter, the coverage focuses on real-time approaches used within industrial networks. With the exception of bandwidth, this includes aspects that are directly related to those approaches. Table 3.1 provides a summary of the coverage.

Depending on the standard, different metrics are used to highlight its timing behavior e.g. cycle times instead of latency. Furthermore, those timing-related metrics are directly affected by the network size and/or cable length in certain standards. On the other hand, the effect of such values is negligible in other standards. Therefore, those metrics were not used, when comparing the standards covered in this chapter. In other words, a direct comparison using timing-related metrics necessitates specific use-case and user requirements.

Table 3.1

	Real-time approach	Communication paradigm	Require time/clock sync.?	Require controller node?	Topology	Bandwidth
AFDX	rate-cons.	event-triggered	NO	NO	star	1 Gbps
CAN	priority	event-triggered	NO	NO	bus	5 Mbps
CAN-aerospace	scheduler	time-triggered	NO	NO	bus	1 Mbps
ControlNet	time-slicing, scheduler	time-triggered	YES	YES	line, tree, star	5 Mbps
EPA	time-slicing, scheduler	time-triggered	YES	NO	line, star, ring, daisy-chain	100 Mbps
EtherCAT	scheduler	time-triggered	YES	YES	star, line, tree, daisy-chain	1 Gbps
Ethernet Power-Link	cycle	time-triggered	YES	YES	star, line	1 Gbps
FlexRay	time-slicing, scheduler	time-, event-triggered	YES	NO	star, bus	10 Mbps
IEEE 1394B	bandwidth reserv., priority	event-triggered	NO	YES	tree	3.2 Gbps
LIN	scheduler	time-triggered	YES	YES	bus	20 kbps
Mil-std-1553C	scheduler	event-triggered	NO	YES	bus, star	10 Mbps
Milcan A	scheduler	time-triggered	YES	YES	bus	1 Mbps
MOST150	bandwidth reserv.	time-, event-triggered	YES	YES	bus	150 Mbps
Profinet IO CC-C/CC-D	cycle, scheduler	time-triggered	YES	NO	star, line, tree, bus	1 Gbps
Sercos III	cycle	time-triggered	YES	YES	line, ring	100 Mbps
TT-Ethernet	scheduler, rate-cons.	time-, event-triggered	YES	NO	star, ring	1 Gbps
TTCAN	scheduler	time-triggered	YES	YES	bus	5 Mbps
TTP/C	time-slicing, scheduler	time-triggered	YES	NO	bus, star	25 Mbps

In certain standards, having a dedicated node within the network, i.e. a controller node, is essential for achieving real-time. Rather than the used real-time approach, this stems from the communication model/method that the network employs, e.g. polling method, producer/consumer model and summation frame method. Even in a scheduler-based approach, the schedule can be configured locally within nodes prior to deployment. In case of dynamic scheduling, changes can be announced and committed based on exchanges between nodes i.e. based on the defined protocol. And conflicts can be resolved using arbitration methods that incorporate properties of the messages being transmitted and/or their senders. Beside the communication method, a dedicated controller node is used to ease network administration and maintenance.

Within the context of time-triggered communication paradigm, network nodes are typically synchronized. As shown in table 3.1, synchronization is not required in CANaerospace. In such standards/networks, a greater emphasis is placed on the quality, i.e. accuracy and precision, of the node's local clock/timer. For example, the jitter/skew of the clock generator/synthesizer chip used within nodes. As such, the minor drifts or shifts in time slots or communication cycles between nodes are considered tolerable i.e. have negligible impact on the overall timing behavior of the network.

As described in sub-section 3.1.4, several standards have been introduced by TSN in order to improve the deterministic behavior of Ethernet. In table 3.1, Ethernet-based standards have introduced relatively minor adjustments to comply with TSN standards, e.g. SERCOS III, PROFINET IO CC-D, AFDX, etc. Those adjustments are minor, because the approaches are mostly similar. For example, AFDX approach can be mapped to ATS traffic shaper on egress ports within the switch. And SERCOS III's traffic scheduling can be mapped to TAS traffic shaper. As for time synchronization, Precision Time Protocol (PTP) implementation , i.e. IEEE 1588 can be replaced by IEEE 802.1AS-Rev.

Chapter 4

Overview of CarRing IV

Before delving into the design and implementation details, this chapter provides an overview of CarRing IV from the point of view of a user. A general and brief description of CarRing IV is provided along with a functional description. In short, this chapter briefly covers what CarRing IV is, how it operates and its possible use-cases.

4.1 General description

CarRing IV can be simply described as a ring-based wired Local Area Network (LAN) that is hard real-time capable, figure 4-1. It is part of a long-term project that is divided into three parts: a protocol, prototype board, and prototype electric car (TUCar). Although I have worked on all three parts, the car or TUCar is outside the scope of this thesis.

4.1.1 Prototype

It is a standalone FPGA-based hardware component with its own memory components and power supply. Two prototypes were developed throughout the project's life-time, a Virtex5-based and a Kintex7-based prototype, check figure 4-2. They include 5- and 12-volt power supplies, respectively.

Regardless of which version, CarRing IV prototypes or nodes are connected to each

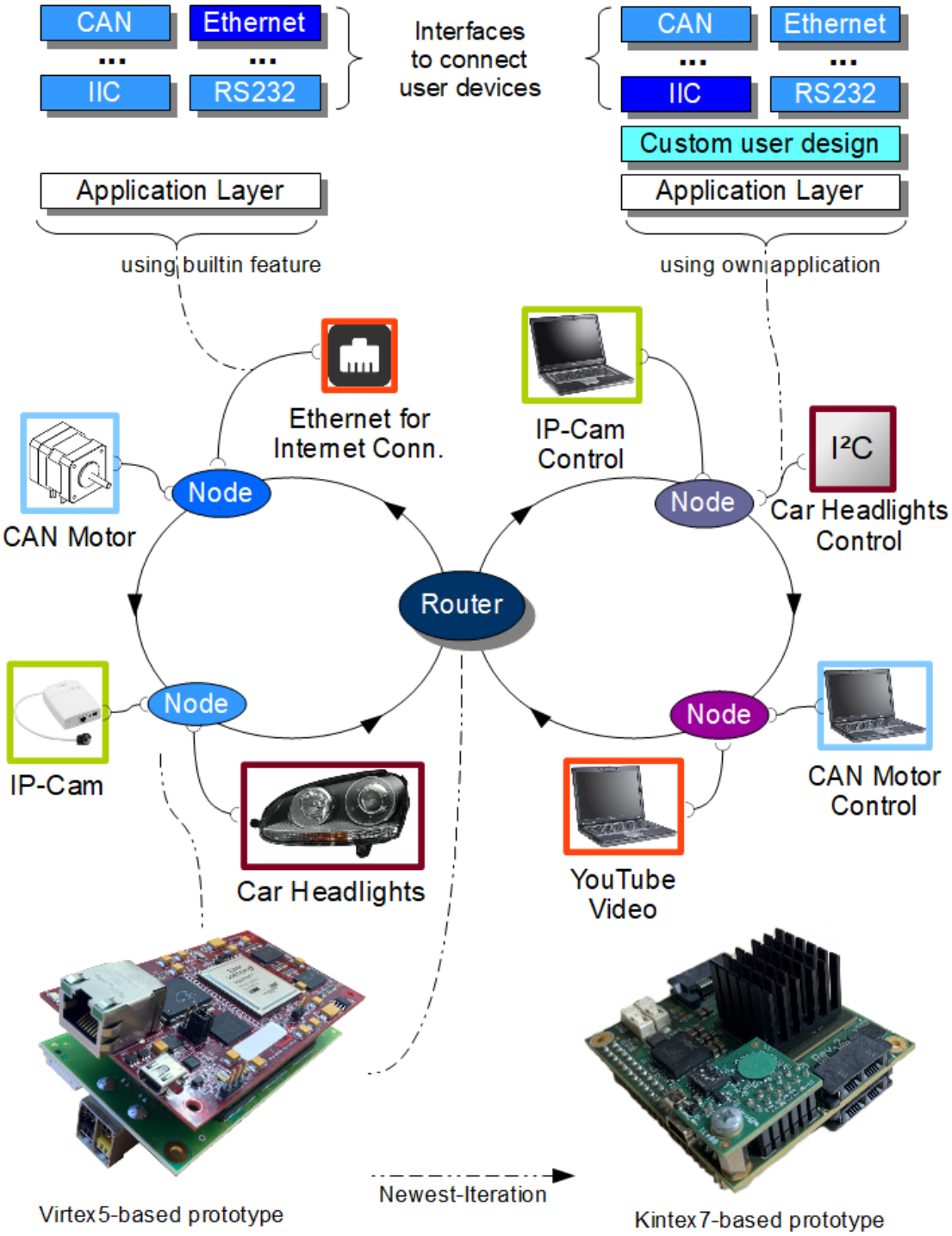
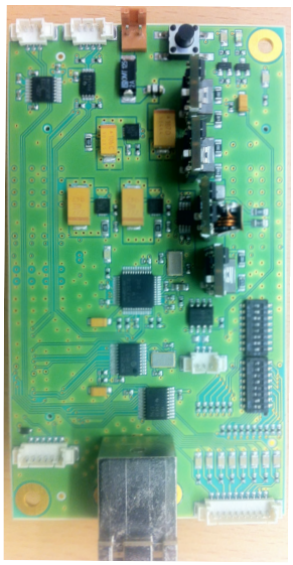
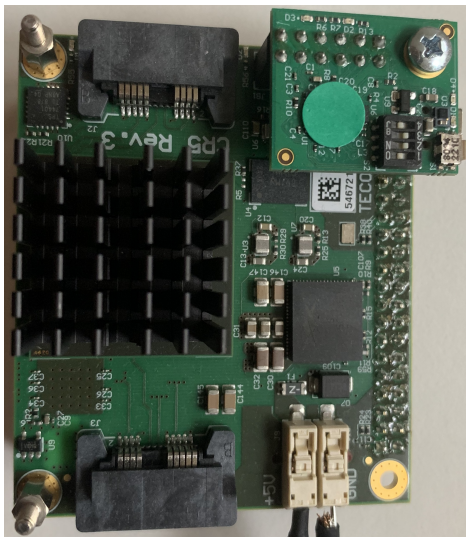


Figure 4-1: CarRing IV and its main components



(a) 101.6 x 57.15 mm



(b) 65 x 56 mm

Figure 4-2: Virtex5-based (a) and Kintex7-based (b) prototypes

other using one physical transmission path, up to 3m cable. In other words, each node has a dedicated point-to-point connection with its predecessor and successor within the ring. This connection is unidirectional or simplex. The entire capacity of the channel is used to send data in one direction. Thus, the data flows in one direction within the ring until it reaches its destination. Additionally, ring topology makes it relatively easier to add or remove nodes. However, this must be done before the network is operational, i.e. when the network is powered-off.

Both prototypes not only have MGTs to transmit protocol data, but also have interfaces that would allow them to transmit data of other networks. In the case of Virtex5-prototype, these interfaces include CAN, RS232, IIC and Ethernet. On the other hand, Kintex7-prototype does not directly include such interfaces, rather it includes intermediary interfaces through which a third-party attachment can be connected. Those intermediary interfaces are RaspberryPi, MikroBUS and PMod. For example, if a CAN interface is required, then a CANberry attachment is connected to the RaspberryPi interface.

4.1.2 Protocol

It is a connectionless hard real-time communication protocol for ring-based LANs. It supports up to 256 rings with 16 nodes per ring. Within each ring, there is one master node that is responsible for initializing it. During ring initialization, all nodes receive their addresses and are permitted to create and push their own frames onto the ring. After initialization, no new frames are created, i.e. the ring has a constant number of frames (3 per node). Those frames are the means by which a node is able to transmit user data. They are continuously circulating the ring regardless of whether the user has any data to send.

While sending user data within a ring, the receiving node copies the frame's payload, update its header and pushes it back onto the ring, i.e. continuous circulation is maintained. As for across-ring transmission, routed packets are not moved from one ring to another until the destination is reached. Rather, only the payload and necessary header information are copied and used to send again using frames that are circulating on the destination ring. In addition to own frames, a node is also permitted to use frames owned by other nodes under certain conditions.

Without delving into the details, hard real-time behavior is achieved due to multiple protocol design decisions. First and foremost is maintaining controlled access to the underlying transmission medium as well as a constant number of continuously circulating frames. Combined with fixed frame size and processing time, the protocol is able to maintain a hard upper limit on latency. Another contributing factor is the removal of prioritization, i.e. no priority field within headers and no priority queues are used. Rather, frame processing is based on arrival time, i.e. first come, first served. Furthermore, fragmentation or segmentation is also removed.

When compared to the OSI model, the protocol includes 4 of the 7 layers, namely physical, data-link, network and application layers. Unlike typical implementation approaches, all layers of the protocol are implemented in hardware (FPGA) using HDLs. Although soft processors can be created and used within an FPGA, no soft or hard processors were used in CarRing IV.

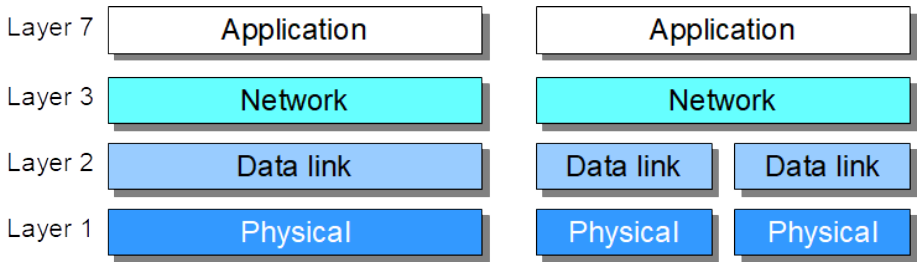


Figure 4-3: Layers in node and router designs

Two implementations of the protocol were created. They allow the prototype to function as either a regular network node or a router. Whether be it Virtex5- or Kintex7-prototype, the router functionality is possible because the prototype has two ports that is split into separate TX/RX interfaces (4), i.e. can be connected to two rings. Therefore, a router has two addresses, one from each ring. Consequently, connecting two rings reduces the overall number of nodes by one. As depicted in figure 4-3, both implementations include all layers.

Unlike typical approaches, the router includes the same layers as a node. This allows the router to maintain the same functionality as a node, i.e. can also be used to send user data. Although each router port has its own physical and data-link layers, they are exact replicas of the corresponding node layers. Beyond protocol layers, the two implementations differ drastically in the amount of FPGA resources used to realize them. Therefore, if a user's requirements can be satisfied with one ring, this would reduce to some extent the power consumption, but more importantly the amount of FPGA resources needed and consequently the cost.

4.2 Functional description

From the user's perspective, the prototype equipped with the protocol is ready for immediate use. As depicted in figure 4-4, its usage can be broadly categorized into direct and indirect. As the name implies, direct use requires the user to introduce their own custom design, written in an HDL, into the prototype's FPGA. In this case, the custom design would use the application layer's interface to send/receive data. Moreover, the additional interfaces, described in subsection 4.1.1, can be used as

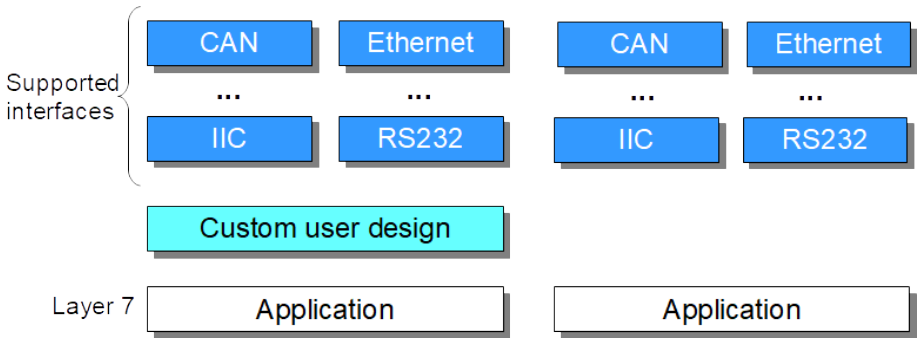


Figure 4-4: Direct and indirect usage of CarRing IV

source/sink for the data being transmitted throughout the network.

On the other hand, indirect use does not require any custom designs or changes on the user's side. Rather, the user simply pushes data through the supported interfaces without interacting with the protocol's application layer. This is made possible using one of the protocol's features, called the transparent mode. Prior to operation, the user simply sets the destination address. During operation, multiple internal custom designs handle data transmission to/from the supported interfaces. Additionally, this feature is also capable of handling multiple supported interfaces simultaneously.

Whether be it direct or indirect use, the protocol's multicast capability allows CarRing IV to emulate field buses, e.g. CAN bus. Moreover, when combining both the transparent mode and multicast, CarRing IV can be used as a backbone for other networks via its supported interfaces. This allows CarRing IV to be easily integrated into existing user systems. In short, user components that are logically connected via CarRing IV are not affected in any respect.

So far, the functionality described above supports event-triggered real-time systems. In such systems, the occurrence of an event triggers data transmission, e.g. the availability of new data on a CarRing IV supported interface. On the other hand, CarRing IV is also capable of supporting time-triggered real-time systems. This is made possible via the protocol's SendAt feature as well as its network-wide clock synchronization. In short, the SendAt feature allows data to be sent at a later point in time, which is provided by the user. It also allows to periodically send the same data based on a time interval provided by the user.

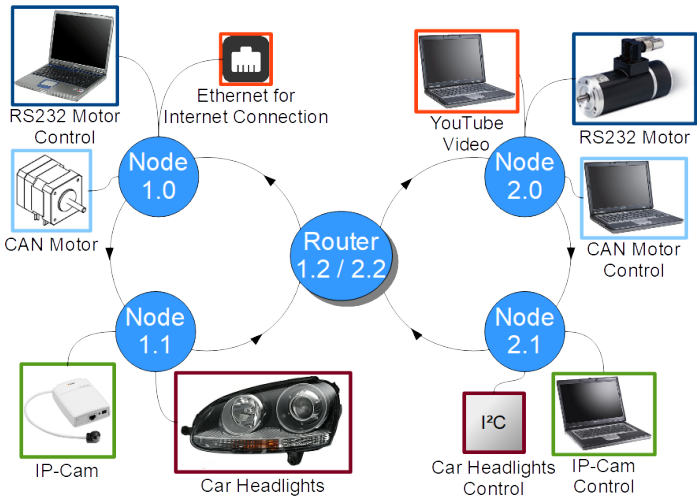


Figure 4-5: Lab demonstration of CarRing IV

4.3 Application Areas

The project's application areas are seen in real-time communication systems within land, air and space vehicles. Whether be it drive- or fly-by-wire, electronic x-by-wire applications are becoming more widespread in both, the automotive and aeronautic industries. Distributed embedded electronic applications are the norm in the automotive industry. There are a variety of such applications with some requiring hard real-time constraints (e.g. braking), while others have less strict constraints (e.g. infotainment). CarRing IV is able to fulfill the requirements of both types of applications. Furthermore, due to CarRing IV high bandwidth, it can also handle traffic generated by both types of applications simultaneously, i.e. they can co-exist and operate in the same CarRing IV network.

The main case-study for CarRing IV is in-vehicle communication. And for that purpose, a lab demonstration was created as a proof of concept and functionality. As show in figure 4-5, both node and router designs are used as well as a variety of supported interfaces per node. This demonstration highlighted the routing capability and the transparent mode feature of CarRing IV. Each node is handling data from multiple interfaces simultaneously and sending the data to another node in the adjacent ring. The numerical values written within the node and router symbols resemble the network-wide address, which is covered in detail in chapter 5.

Chapter 5

CarRing IV Layers

After providing an overview of CarRing IV in chapter 4, this chapter describes the protocol and its layers in detail. Each layer is covered in its own dedicated section below. Prior to describing the specifics of each layer, in what follows, the non layer-specific information is covered, such as topology or aspects in which the protocol and its design differs from typical or standard approaches. Moreover, there are protocol features that incorporate multiple layers to operate. Those have been moved to their own dedicated chapters, namely network-wide clock synchronization and reliable multicast.

CarRing IV is a ring-based LAN that is capable of hard real-time communication. A ring allows for bidirectional communication between network nodes using unidirectional wires, thereby reducing cable length. A CarRing IV network is a set of coupled physical rings as opposed to logical ones. In the latter, a node does not have a dedicated and direct point-to-point connection, i.e. a physical transmission path, with its predecessor and successor within the ring. For example, a star ring topology where a central hub acts as connector, or a bus ring topology where all nodes are connected to one cable, i.e. bus. On the other hand, if one node fails in a physical ring topology, then the network's operation is halted. Such a problem can be solved by standard remedies, such as implementing a physical bypass within each node or via hardware redundancy, e.g. using a dual ring topology where the secondary ring acts as a backup or reserve.

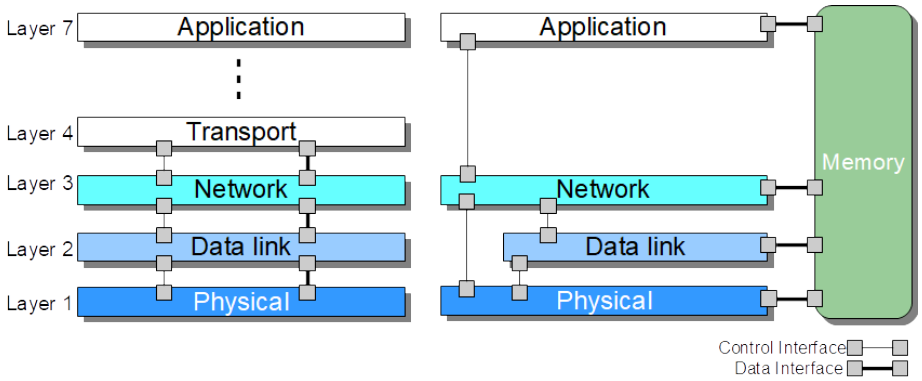


Figure 5-1: Typical inter-layer data-exchange vs CarRing IV

Up to 256 rings, with 16 nodes each, can be created in a CarRing IV network. Those rings are interconnected using a CarRing IV router. It can connect two rings to one another and therefore, it has two addresses, one from each ring. Unlike typical router designs, it has the same number of protocol layers as a regular CarRing IV node. In other words, the router retains the same functionality as a node, i.e. can be used to transmit user data. Consequently, connecting rings together reduces the total number of nodes within a network which in turn, reduces power consumption and network cables required.

Whether be it a node or router, they both implement the CarRing IV protocol. It is a connectionless hard real-time communication protocol that consists of 4 layers, namely physical, data-link, network and application. Although the protocol layers are based on the OSI model, the protocol design does not strictly adhere to it. The main two reasons are, reducing processing time which affects latency as well as reducing the amount of FPGA resources required to implement the protocol.

As shown in figure 5-1, the physical layer interacts with and is fully aware of the frame and packet formats of both the data-link and network layers. The corresponding inter-layer data exchange is done via shared memory, where each layer has its own address range. Memory management and the associated design components are covered in detail in chapter 8.

Within the context of protocol layering, encapsulation and decapsulation performed in CarRing IV, differs from typical approaches where each layer has its own

header and possibly trailer. The protocol's data-link and network layers still have separate headers each with their own fields. However, those headers are merged into one during multicast which is covered in chapter 7. The new header is still processed by both layers for different purposes. Whether be it a merged or two separate headers, the overall number of bits allocated for headers remains fixed, while the internal structure differs based on the frame's type. This is possible because frame processing in the protocol is designed and implemented to always start with the type field of the frame's header.

In order to maintain a deterministic behavior, the protocol uses a fixed size frame, i.e. a fixed payload and header(s) size despite variable internal structure. In addition to the measures described in the following sections and having a deterministic behavior, achieving hard real-time communication is also made possible by deliberately removing protocol features that negatively impacts latency, i.e. by removing any form of segmentation, fragmentation or prioritization. More specifically, no segmentation or fragmentation by a sender node or intermediate router(s). And there is no priority field(s) within headers and no priority queues when sending or routing user data.

In other words, all user traffic within the network has the same priority, i.e. hard real-time requirement is assumed for all traffic. Thus, avoiding starvation condition which is present in priority queuing. Despite having a fixed frame size, using weighted fair queuing would still introduce delays in user data delivery. Nevertheless, CarRing IV use is not limited to systems with hard real-time requirement. Due to its high bandwidth, it can support and be used for systems that have firm, soft and non-real-time requirements.

Last but not least, the protocol also includes measures that limits the effects of a misbehaving user application to its own node within the ring. Those measures are implemented in the data-link and application layers. In short, a user application is neither allowed to create and introduce new frames into the ring, nor is it allowed to indirectly block the processing of new incoming frames within a node. The latter is possible if a user application either delays or does not read a received payload from memory. In other words, if a user application misbehaves or attempts to exceed the

allocated resources for its node, then it would only affect its own node and it will not disrupt the timing of other nodes and consequently other applications.

5.1 Physical layer

5.1.1 Overview

Similar to other protocols, the purpose of the CarRing IV's physical layer is as stated in the OSI model [21]. However, since this body of work also includes a hardware implementation, the layer description below includes detailed information that are influenced by the underlying prototype. As described previously, two prototypes were developed throughout the project's lifetime, Virtex5- and Kintex7-prototype. Although no longer current, Virtex5-prototype is still covered due to the valuable insight gained with respect to the protocol's behavior and measurements, which are described in chapter 9.

CarRing IV's physical layer uses digital transmission coupled with a block coding technique, 8B/10B encoding [41] for both the previous Virtex5-prototype and the current Kintex7-prototype. Using a dedicated physical transmission path, the physical layer transmits the protocol's data sequentially from one node to another within the ring. It uses synchronous serial transmission, where serial data is transmitted in differential Non-Return-to-Zero (NRZ) format.

The only layer that can be replaced is the physical layer with the condition that the replacement is able to match the requirements and provide the same services as the original. Those services are covered in the section below. As for the requirements, it must have a deterministic behavior and supports a framing interface. More specifically, the amount of time to send a frame from one node to its successor within the ring must be fixed. This can be achieved by using a dedicated physical transmission path, where the entire capacity of the corresponding channel is dedicated to transferring data between two nodes.

As for the framing interface support, this is separate and apart from and not to

be confused with the protocol's frames. In short, data can be transferred between two nodes either as a stream of data or framed data. The latter adds delimiters to start and end of the data being transferred. Using a streaming interface implies that the protocol must introduce its own delimiters to distinguish between two consecutive frames.

5.1.2 Services provided to higher layers

Similar to OSI model, the layer provides:

- Point-to-point physical connections, where each node has a unidirectional physical transmission path to its successor and predecessor within the ring.
- Preserve the transmission order of bits, whether be it Most Significant Bit (MSB) to Least Significant Bit (LSB) or vice versa.
- Error detection and notification of a corrupted frame.
- QoS parameters i.e. error rate, transmission rate and transit delay.

Unlike typical approaches, incoming data is not pushed from one layer to another, rather the physical layer is responsible for encapsulation/decapsulation for both the data-link and network layers. Moreover, it pushes the necessary information for both layers to initiate processing of the incoming frame/packet. This includes the memory addresses of the headers and payload.

Last but not least, CarRing IV prototype can be connected up to only two rings. Therefore, in case of a router design, the physical layer is not required to provide explicit identifiers for the underlying ports.

5.1.3 Sub-layers

The physical layer consists of 3 sub-layers, as depicted in figure 5-2. While encoding, serialization and transceiver are beyond the scope of this thesis, only relevant information of both sub-layers are covered. This information is crucial to the understanding

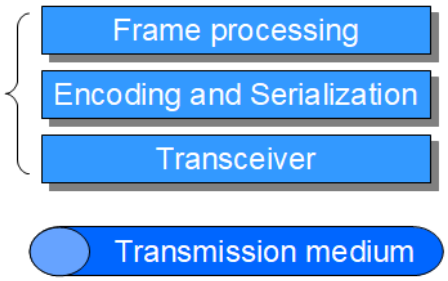


Figure 5-2: Sub-layers of CarRing IV's physical layer

of the protocol's behavior and performance. In what follows, a brief description of each sub-layer as well as the transmission medium is provided.

Frame processing

It encapsulates/decapsulates the protocol's headers and payload. As well as, reading/writing protocol frames from/to shared memory. Using LocalLink (LL) interface [39], it pushes protocol data to Aurora.

Encoding and Serialization

It is responsible for the encoding and serialization of the protocol's data. This data is received from the above layer in chunks i.e. smaller parts. The size of each chunk of data directly affects the amount of time required to send the protocol's frame. The size depends on the properties of the underlying transceiver. Xilinx's Aurora 8B/10B [43] is used for this sub-layer.

Transceiver

As the digital data is converted to digital signals, transceiver transmits them to the next node using the underlying transmission medium. It consists of one or more physical lanes. The number of lanes affects the size of data chunks described above. The transceiver used is Xilinx's GTX transceiver [40] [42].

Transmission medium

To support high-speed transmission, i.e. MGTs, network cables that have high bandwidth were used. In case of Virtex5-prototype, shielded twisted copper cables of category 7e. As for Kintex7-prototype, Twinaxial or "Twinax" cables were used.

5.1.4 Functional description

As described previously, the physical layer is responsible for encapsulation/decapsulation and reading/writing the corresponding headers and payload into shared memory. Figure 5-3 provide a general depiction of the memory allocations and operations involved in sending/receiving of a frame. The description below follows the same structure of the sub-layers section.

Frame processing

While sending a frame, this sub-layer only interacts with the data-link layer. As shown in figure 5-3, the shared memory is composed of six logical areas with read/write interfaces. Within each area, memory address offset is the means by which headers and payload are correlated, i.e. form a frame/packet. To distinguish between the sender and receiver side address offsets, additional information is provided along with the address offset. This is covered in more detail in chapter 8.

After receiving the address offset, it uses the appropriate memory read interface to encapsulate and pushes the frame to the next sub-layer. As shown in figure 5-3, the memory address offset could be pointing to a frame that is carry user data or a frame that is being forwarded, i.e. belongs to another node. Regardless of the source, after the frame is pushed to the next sub-layer, the frame processing sub-layer signals that the corresponding memory allocations can be reused for a new transmission.

On the other hand, the receiver side interacts with both the data-link and network layers. In order to receive an incoming frame, a memory address offset must be first fetched. Unlike the sender side, the receiver side is responsible for acquiring it. This is aided by one of the memory management design components.

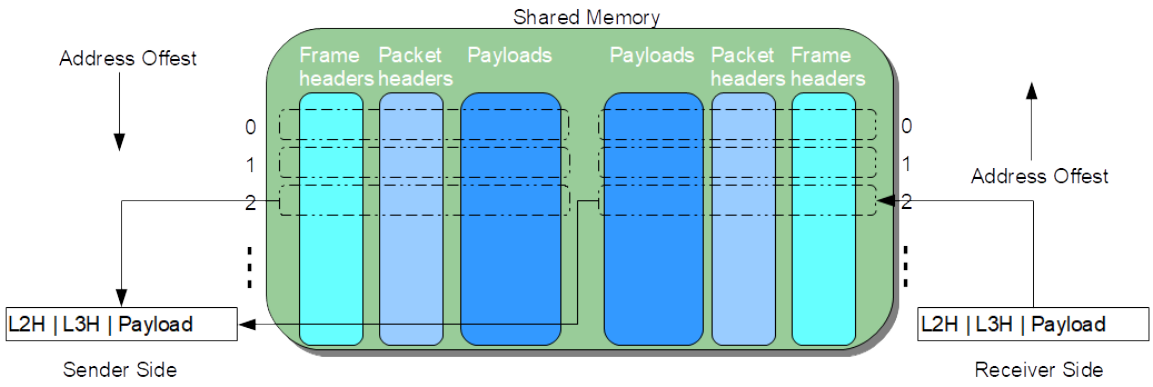


Figure 5-3: Overview of encapsulation/decapsulation and memory operations

As soon as each header is received, the corresponding address offset and processing information is immediately pushed to the appropriate layer. This allows higher layers to initiate processing while the payload is being received. Thus, any time consumed by processing that overlaps with receiving the payload is effectively masked. Depending on the destination of the incoming frame/packet, the corresponding memory allocations are signaled for reuse by either the receiver side of the application layer or the sender side of this sub-layer.

Encoding and Serialization

This sub-layer is implemented using Xilinx's Aurora 8B/10B [43]. While the internal workings of Aurora are beyond the scope of this thesis, some of its aspects and properties have direct implications for the protocol's implementation and performance. However, it does not affect the protocol's deterministic behavior. In what follows, those implications and the associated properties are covered.

Whether be it sending or receiving a frame, the LL interface is used to exchange data between Aurora and the frame processing sub-layer. Since the protocol is implemented in hardware, processing time is primarily expressed in clock ticks or clocks as opposed to seconds. Clocks are covered in more detail in chapter 2. Similar to any data interface, LL can accept only a limited number of bits per clock. This is referred to as the data-width of the LL interface. Thus, the data-width of LL affects the number of clocks, i.e. time, required to push or receive a frame to/from Aurora.

In turn, the data-width can not be set arbitrarily. Rather, it depends on the customization and implementation of Aurora. More specifically, it depends on the number of lanes in the underlying transceiver sub-layer as well as the number of bytes per lane set within Aurora. Aurora permits either 2 or 4 bytes per lane. In short, data-width is equal to number of lanes multiplied by the number of bytes per lane.

Other than the number of clocks required to push a frame to Aurora, the number of bytes per lane directly affects the transceiver latency. In other words, it affects the number of clocks required for the first chunk of data sent to appear on the receiver side of the adjacent node. The higher the bytes per lane, the more clocks are required to send a chunk of data between two adjacent nodes.

Transceiver

Another effect of setting 2 or 4 bytes per lane is the period-value of the design-clock outputted by Aurora. Fewer bytes per lane results in a smaller design-clock period. As described in chapter 2, in order for an RTL hardware design to operate successfully within an FPGA, its timing constraints must be matched. Each RTL hardware design has a lower limit on clock period beyond which it would fail its timing constraints. In other words, it would not operate successfully within the FPGA.

On the other hand, to allow any meaningful processing in the protocol's higher layers, the first chunk of data received from Aurora must include the frame's header. Thus, LL's data-width must be large enough to include the frame's header which is 24-bits in size.

In the Virtex5-prototype, the transceiver has a total of two lanes. In order to accommodate a router design that connects two rings, only one lane is used to transmit data in a ring. Therefore, Aurora is set to 4 bytes per lane, i.e. 32-bits data-width. And the corresponding transceiver latency was 61 clocks. The resulting design-clock period (12.8 ns) is within the limits of the protocol's RTL hardware design.

As for Kintex7-prototype, the transceiver has 8 lanes. Similarly, to account for a router design, 4 lanes per ring can be used. Although setting 2 bytes per lane would amount to 64-bits data-width, the corresponding clock period (3.2 ns) is inadequate

for the protocol's RTL hardware design. Therefore, Aurora is set to 4 bytes per lane, i.e. 128-bits data-width and the resulting clock period is 6.4 ns. As for the transceiver latency, its 41 clocks.

Last but not least, the combination of Aurora's protocol engine and transceiver latencies causes a pipeline effect in data transmission. In other words, while the first chunk of data is being transmitted, Aurora still accepts new chunks of data as long as its internal buffers are not full.

Transmission medium

Each of the aforementioned transceiver lanes consists of TX and RX parts. Those parts are physically connected to ports within the prototype. In order to form a physical ring topology, the prototype has a total of four ports. Each port is connected to either the TX or RX parts of one or more lanes. In the case of Virtex5-prototype, each port connects to either the TX or RX parts of one lane. As for Kintex7-prototype, each port connects to either the TX or RX parts of four lanes.

In other words, the transceiver lanes and the underlying physical ports are divided into 2 TX and 2 RX interfaces. Thus, four cables are used to connect those four interfaces, where each network cable forms a unidirectional physical transmission path between two nodes.

5.1.5 Error detection and recovery

In CarRing IV protocol, error detection and recovery is not required at each layer. The protocol is designed to be used in its entirety, i.e. not partially where one layer is extracted and used in a different protocol suite, e.g. TCP/IP protocol suite. Furthermore, the physical layer is the same for all nodes/routers within a CarRing IV network. Therefore, error detection and recovery is performed once on the physical layer. As opposed to networks with varying physical layers which requires error detection to be performed on the next identical layer, e.g. data-link layer.

The 8B/10B encoding inherently includes error detection capability, single-bit

and most multi-bit errors [43]. In addition to the built-in error detection, Aurora also includes 32-bit Cyclic Redundancy Check (CRC). Along with the LL interface, Aurora also includes an interface that propagates any errors detected to the frame processing sub-layer. Using Aurora's interface, the frame drop count is propagated upwards through layers as statistical information which can then be used by the network administrator. Based on prolonged tests of the prototypes, no frame errors where encountered. Those tests included frame generator and checker RTL hardware design components where an ongoing data transmission was carried out for up to three full days without pause or break.

If encountered, any error frame is immediately dropped. The dropped frame is automatically detected, recreated and reintroduced into the ring by the data-link layer of the node to which the frame belonged.

Nevertheless, if there is a constant source of error either due to the network's environment, change of transmission medium (e.g. interference in wireless connection) or the protocol is applied to another different prototype, then error correction must be used, e.g. Punctured Turbo Convolutional Coding (PTCC).

5.1.6 Router vs Node design

CarRing IV routers do not have a dedicated prototype. However, they do have a dedicated design. So far, the descriptions provided above apply to the node design. In order for a router to connect two rings, all prototype physical ports are activated and used to transmit frames, while in a node design only half are active.

In the router design, each ring that the router connects to, has its own physical layer and own memory allocations, i.e. two physical layers. Those two layers are essentially duplicates of the physical layer in the node design. Therefore, the number of routers within a ring does not affect its timing, i.e. the deterministic behavior is maintained.

The only difference between router and node physical layers is the sender side of the layer. Within a router, each layer's sender side is able to access the memory allocations of the other physical layer. Thus, reducing the overhead, i.e. processing

time, associated with routing.

5.2 Data-Link layer

5.2.1 Overview

As described above, the physical layer exhibits a deterministic behavior while transmitting data from one node to another. While each node has a dedicated direct point-to-point connection to its successor and predecessor within the ring, the connection is unidirectional. In other words, each node must also transmit frames that are destined for other nodes. This effectively makes the underlying transmission medium a shared resource between all nodes within the ring. And each node has a secondary function similar to that of a repeater. More specifically, frames received from the predecessor node are sent again to the successor node, i.e. pushed back again into the ring.

To maintain the deterministic behavior of the protocol, access to the underlying transmission medium, i.e the shared resource, must be controlled. This is the main purpose of the data-link layer. It is also capable of detecting dropped frames in the physical layer. Such dropped frames are recreated and reintroduced into the ring by the data-link layer.

Although both unicast and multicast transmission are supported, broadcast transmission is excluded from the protocol. It is considered as a special case multicast transmission, where all nodes within the network belong to the same multicast group.

Due to data-link layer's SendAt feature, CarRing IV is also able to support not only event-triggered systems, but also time-triggered ones. In short, the feature allows periodical transmission of the same frame based on a specified time interval as well as delayed transmission using a specified time value. Both the time value and interval are given by the user.

In addition to node and router, the data-link layer adds another functional distinction, primary and secondary. Only one primary or primary node is allowed per ring.

In short, it is responsible for initializing the ring, which includes assigning addresses and distributing other metrics that are required for network operation. Since a router has the same functionality as a node, it too can be a primary. All other nodes and routers within the ring are considered secondary. Secondaries do not have special functionality. Last but not least, transmitting user data across rings is delegated to the network layer. In other words, there are no data-link layer switches in CarRing IV. Rather, a router that is capable of connecting two rings.

5.2.2 Services provided to network layer

When compared to the OSI model, the data-link layer of CarRing IV only supports connectionless mode and provides the following services:

- Data-link addresses where two types of addresses are supported, unicast and multicast.
- Framing using fixed size frames that include a header, but neither include nor require a trailer. While this service is provided by the data-link layer, the encapsulation/decapsulation task is delegated to the physical layer as described previously.
- Both acknowledged and unacknowledged data transmission.
- Acknowledgment notification. In the case of Acknowledgment, a node or router is able to confirm (with a positive acknowledgment) or negate (with a negative acknowledgment) the reception of the corresponding frame's payload. In addition to positive and negative acknowledges, the data-link layer also notifies the upper layer if no acknowledges are received within a certain time limit.
- Destination found notification. In both acknowledged and unacknowledged data transmission, the data-link layer notifies whether the destination was found.
- Reliable multicast. This is covered in detail in chapter 7.
- SendAt which allows periodical and delayed transmission of a frame.

- Acquiring and providing the node's or router's network-wide address.

Last but not least, the data-link layer does not include error detection, notification and recovery. As previously described, such tasks are delegated to and performed once at the physical layer.

5.2.3 Access control

The access method used by the data-link layer of CarRing IV falls under the category of controlled-access protocols. This is in contrast to random-access and channelization protocols, where they utilize collision detection and avoidance techniques, or some form of frequency, time or code division techniques.

After power-up, i.e. at the beginning of network operation, each node creates and introduces its own frames into the ring. Regardless whether they are carrying user data, those frames are continuously circulating the ring. Each node must create only a fixed number of frames (3). Once all nodes create their own frames, no new frames are created during network operation. Furthermore, no new nodes can be introduced into the network after power-up. This results in a fixed number of continuously circulating frames, which in turn resemble a shared resource used by all nodes within a ring.

CarRing IV has two modes of operation, isochronous and anisochronous. The operation mode can not be altered during network operation, i.e. it is set prior to power-up. In isochronous mode, a node can send user data using only its own frames. In other words, the node must wait for one of its own frames to circulate back to it and use that frame to send user data. On the other hand, the anisochronous mode allows a node to use frames owned by other nodes to send user data. This mode defines conditions and scenarios in which such frame-reuse is permitted.

In essence, the access method is quite similar to the reservation method within the category of controlled-access protocols. Rather than requiring each node to make a reservation before sending user data, a fixed number of reservations are made automatically at the beginning of network operation. Thus, deterministic behavior is

achieved due to the fixed number of circulating frames and having a fixed size frame. This allows CarRing IV to have a hard upper limit on latency.

With respect to timing behavior, both modes are classified as hard real-time. As described in section 3.1, isochronous real-time maintains a more strict timing behavior than abiding by a hard upper limit. While being more deterministic, this negatively impacts performance.

Within the context of CarRing IV, the isochronous mode offers higher precision, while anisochronous offers higher performance. The term precision refers to lower variance in latency. Since the anisochronous mode enables frame-reuse, the corresponding performance metrics are impacted by the level of activity of other nodes within the ring. In other words, the fewer overlapping/simultaneous user send requests, the higher frame-reuse and consequently performance is achieved. In case of the highest possible per-node user send requests, the difference between the two operation modes is greatly diminished. Within the context of performance metrics, the anisochronous mode's lower limit is that of the isochronous one.

Last but not least, CarRing IV's access method does not require network-wide clock synchronization. Nevertheless, as described in chapter 6, such a synchronization is introduced and implemented with the main purpose being the extension of CarRing IV's support for time-triggered communication.

5.2.4 Addressing and frame format

Two types of addresses are supported by the protocol, unicast and multicast addresses. The latter is covered in chapter 7. As described previously, broadcast addresses are considered a special use-case of multicast addresses and are therefore excluded.

The data-link address, also referred to as MAC address, is unique within the ring. It is 4-bits in size and presented as a decimal digit, i.e. 0 to 15. Therefore, a ring can have up to 16 nodes. It is assigned automatically by the primary node during ring initialization. Starting with 0 for the primary node, MAC addresses are assigned with (+1) increments for the secondaries throughout the ring. In other words, the MAC address is based on the node's position within the ring and whether the underlying

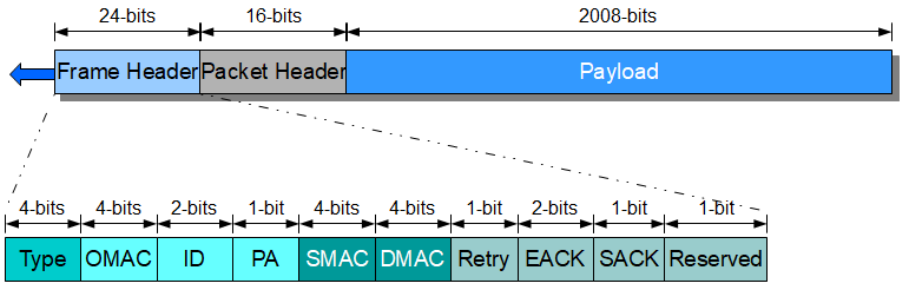


Figure 5-4: Generic Frame format

unidirectional wired connections reflect a clockwise or counter-clockwise flow of data.

The data-link layer uses frames of fixed size. The format of such a frame is depicted in figure 5-4. Since frame errors are handled in the physical layer, no trailer is required. Trailers are typically used for some form of checksum or CRC. As for the header, in what follows, a brief description of each field is provided.

- **Type**. It is a 4-bit field that defines the type of data that the frame is carrying. Some frame types can be used to carry user data, while others are used internally by the protocol.
  - `Reset`. Only the primary can use this frame type. It is used during ring initialization i.e. at the beginning of network operation. It is for protocol internal use only.
  - `Transport`. After ring initialization, this is the default type, when no user data is being transmitted. It is for protocol internal use only.
  - `Ethernet Service`. In case of unacknowledged data transmission, the user can set this type.
  - `Acknowledged Point To Point`. It is set by the user to indicate that an acknowledge is required to confirm data reception.
  - `Request`. Similar to the type above, it can be set by the user and requires an acknowledge. Additionally, it indicates that a response is required from the receiver.
  - `Response`. This is the counterpart of the Request type above. It also requires an acknowledge.

- `Positive Acknowledge`. It is only used internally by the protocol in association with the `Acknowledged Point To Point`, `Request` and `Response` types.
- `Negative Acknowledge`. It is the counterpart of the positive acknowledge type. Both types can not be set by the user.
- `Multicast`. It is used for multicast transmission, which is covered in chapter 7.

- **Owner MAC (OMAC)**. Each node creates and introduces its own frames into the ring. This 4-bit field holds the MAC address of the owner node. After ring initialization, its value is fixed i.e. it is read-only.

- **Frame identifier (ID)**. Since each node creates three frames, this 2-bit field is used to distinguish those frames from one another. Its value is also fixed i.e. read-only.

- **Public Access (PA)**. It is a 1-bit field that is used as part of the public access feature. In short, the feature allows other nodes to also the use frame when the owner node has no user data to transmit.

- **Source MAC (SMAC)**. Since frames can also be used by other nodes, this 4-bit field also holds MAC address of the source node that is sending the user data.

- **Destination MAC (DMAC)**. It is a 4-bit field that holds the destination MAC address for the user data being transmitted.

- **Retry**. Although not ideal, the protocol allows for acknowledged data transmission to be retried using this 1-bit field.

- **Embedded Acknowledge (EACK)**. This 2-bits field is used as part of the i-to-j feature.

Figure 5-5: Typical sub-layers of data-link vs of CarRing IV

- **Skip Acknowledge (SACK)**. During an acknowledged data transmission, this 1-bit field is used to explicitly indicate to the data-link layer of the receiver that no positive or negative acknowledges are required.

Although figure 5-4 reflects the payload size, the size changed based on which prototype was used. However, the protocol timing was not affected despite those changes. In short, the frame in its entirety required a fixed 16 clocks to be pushed to Aurora using its LL interface. The changes as well as the corresponding reasoning behind them are covered in chapter 9.

Last but not least, frames have a fixed size, but a variable internal header structure. Figure 5-4 depicts a generic frame header structure which applies to most of the frame types i.e. `Transport`, `Ethernet Service`, `Acknowledged Point To Point`, `Request`, `Response`, `Positive Acknowledge` and `Negative Acknowledge`. As for `Reset` and `Multicast`, those are covered in section 5.2.6 and chapter 7, respectively.

5.2.5 Sub-layers

Typically, the data-link layer is divided into two sub-layers, Data Link Control (DLC) and MAC. The upper sub-layer, DLC, can have either a connectionless or connection-oriented protocol. It provides services such as framing, flow and error control. While the lower sub-layer, MAC, controls and coordinates the access to the link, i.e. underlying physical layer and transmission medium.

As show in figure 5-5, the data-link layer of CarRing IV is also divided into two sub-layers. However, the purpose and functionality of those sub-layer differ from the typical approach. A brief description of each sub-layer is provided below.

ASA

The protocol supports acknowledged data transmission, where the receiver node sends back a positive or negative acknowledge to the sender node. The Acknowledgment and SendAt (ASA) sub-layer is responsible for tracking and handling those acknowledgments as well as retrying the transmission if either a negative or no acknowledge is received within a certain time limit. The latter is attempted only if its explicitly requested by the user. Moreover, it is also responsible for retransmission in case of multicast, which is covered in chapter 7.

In addition to acknowledgment, this sub-layer also includes the SendAt feature. In short, this feature enables CarRing IV to support time-triggered systems by providing periodical and delayed data transmission capabilities.

ACF

The Access Control and Framing (ACF) sub-layer is more or less the equivalent of both the DLC and MAC sub-layers described previously. In addition to providing the framing service and controlling access to the underlying link, it is also responsible for node initialization. And in the case of the ring's primary node, it is also responsible for ring initialization. In short, initialization includes acquiring a node's address and other metrics crucial for the network's operation.

5.2.6 Functional description

This section is structured based on network operations encountered during run-time. Other than ring initialization and data transmission, this section also covers the SendAt, public access and i-to-j features.

Ring Initialization

As the name implies, this is the very first network operation which takes place immediately after power-up. During ring initialization, only the primary can send frames, while secondaries passively wait. Within the data-link layer, initialization is han-

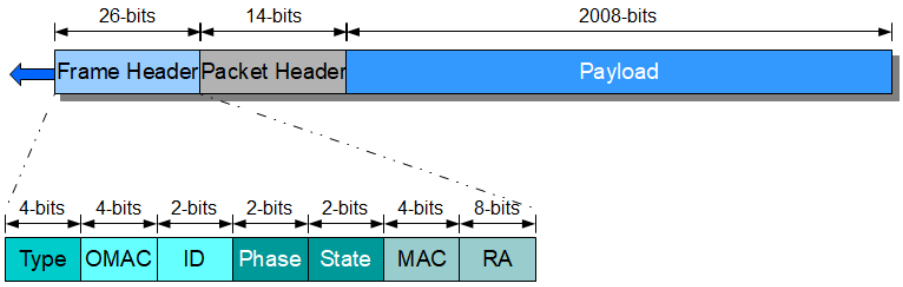


Figure 5-6: Reset Frame format

dled by the ACF sub-layer. It is unaffected by which operation mode is used, i.e. isochronous or anisochronous.

Ring initialization is a multi-step process that is divided into three phases. Those phases are accomplished sequentially, i.e. one phase must be finished before the next one can start. Therefore, the primary uses only one frame of type `Reset` to initialize the ring. In what follows, this frame is referred to as the Reset frame. As described previously, while maintaining an overall fixed header size, the internal header structure depends on the frame's type. Figure 5-6 depicts the header structure of a reset frame. Prior to describing each phase within the ring initialization process, a brief description of each new header field is provided below.

- **Phase**. This 2-bits field indicates which phase of the ring initialization process is engaged, i.e. bits reflects a value of 0 for `Assign address`, 1 for `Circulation period` or 2 for `Clock synchronization`.
- **State**. It is 2-bits that is used as part of the network-wide clock synchronization, which is covered in chapter 6.
- **MAC**. Similar to OMAC, it is a 4-bits field that is used to set the MAC addresses of secondaries.
- **Ring Address (RA)**. It is an 8-bits field that is used to set the network layer addresses of secondaries.

Phase One - Assign Address. Two tasks are accomplished during this phase, assigning addresses to secondaries as well as calculating the circulation period. The

primary starts by creating a reset frame. In addition to setting the type field to `Reset`, the phase field is set to `Assign address` and MAC address field is set to 0. As for the ring address field, it is set based on the configuration data within the prototype. In short, unlike MAC addresses, ring addresses are configured statically by the user in the primary node. Thus, the assign address phase not only sets a node's MAC address, but also its ring address i.e. network layer address.

As soon as the reset frame is sent, an internal counter is started within the primary. Once the reset frame circles back to the primary, the counter is stopped and its value is used as the circulation period. In other words, the circulation period reflects how many clocks are required to traverse the ring. This value is used by each secondary when creating their own frames as well as for network-wide clock synchronization.

When a secondary receives the reset frame, it reads the MAC header field and increments its value by one. In addition to the node's MAC address, the resulting value is also used to update the MAC field before sending the reset frame to the next node in the ring. As for the ring address field, it is copied and used as the node's ring address. The combination of both addresses constitute the node's network-wide unique address.

Phase Two - Circulation Period. In this phase, the primary distributes the circulation period calculated in the previous phase. When the reset frame reaches the primary, its phase field is update to `Circulation period`. And the circulation period is set using the frame's payload. The secondaries simply copy the circulation period in the payload and send the reset frame to the next node in the ring.

Phase Three - Clock Synchronization. The last phase is part of the network-wide clock synchronization. The state header field is used in this phase, which is covered in chapter 6.

As soon as the last phase is finished, the primary deletes the reset frame. Thereafter, the primary as well as each secondary is allowed to create and introduce their own frames into the ring. In order to avoid burst-like behavior, when transmitting user data, a node does not create and introduce its three frames into ring one after the other, i.e. consecutively. Rather, an artificial gap is used between frames. Its value

is equal to the circulation period divided by three. In other words, each node waits (CirculationPeriod/3) clocks before creating and introducing the next frame into the ring.

Furthermore, this artificial gap also accounts for the scenario, where the user might not be fast enough to use all node's frames consecutively. In what follows, the ability to use a frame to send user data is referred to as a sending opportunity. Since the anisochronous mode allows nodes to use frames owned by others, it has more sending opportunities than the isochronous mode.

Last but not least, the newly created frames use the `Transport` type. It is the default type, when there is no user data being transmitted. In other words, once a user data transmission is concluded, the corresponding frame's type is always updated to `Transport`. In what follows, such a frame is referred to as Transport frame.

Unacknowledged data transmission

This type of data transmission is reserved for scenarios where confirmation is not required upon data reception. For that purpose, the `Ethernet Service` frame type is used. Once the frame's payload is copied into the receiver's memory, the receiver can immediately update the frame type to `Transport` and send the frame back into the ring.

Within the data-link layer, it is handled by the ACF sub-layer. This includes acceptance criteria and frame processing.

Acknowledged data transmission

Unlike the previous type of data transmission, this one requires a confirmation from the receiver node. It is intended for scenarios, where increased reliability is required. Furthermore, it also includes a limited support for retransmission, which in turn is triggered by either receiving a negative acknowledge or no acknowledge within a certain time limit, i.e. timed-out. This time limit is based on the circulation period described above. Since retransmission negatively impacts latency, it is not performed automatically, rather it must be explicitly requested by the user, i.e. part of the user's

send request.

In order to maintain a hard upper limit on latency, acknowledgments are only supported within the sender's ring, i.e. they do not traverse rings. In other words, beyond sender's own ring, the data transmission is treated as an unacknowledged one. In such a scenario, the first intermediate router, i.e. within the sender's ring, still sends an acknowledge back to the sender. However, all other intermediate routers as well as the receiver do not send back an acknowledge to the sender or any intermediate router along the path.

Within the data-link layer, the acceptance criteria remains in the ACF sub-layer, while frame processing is split between both sub-layers. ACF handles the frame transmission. And the ASA sub-layer is responsible for handling the acknowledgments and retransmission. If retransmission is required, ASA reissues the send request that was originally made by the user. From the ACF perspective, it looks like another user send request.

Positive and negative acknowledges are not issued directly by the user. Rather, they are issued internally by the protocol. They are based on whether the user on the receiver end is able to read the frame's payload from memory in time. Since nodes also function as a repeater where they are constantly transmitting other frames, a slow or malfunctioning user application might otherwise block data transmission, which in turn negatively impacts the network's timing.

Unlike typical approaches, acknowledges in CarRing IV are immediate. The same frame that is carrying user data is used to carry the returning acknowledge. In other words, no delayed acknowledges are allowed, where a separate frame is required to send an acknowledge. This is referred to as an `Immediate Acknowledge`. And it eliminates the need for `Sequence Numbers`, which is typically required for handling acknowledgments. In case where retransmission is explicitly requested, acknowledges are tracked and processed using a frame's `OMAC` and `ID`, which remain constant i.e. read-only header fields. The combination of those two values uniquely identifies frames circulating within the ring.

Last but not least, five frame types are used in acknowledged data transmis-

sion. Returning acknowledges use either the `Positive Acknowledge` or `Negative Acknowledge` frame types. While `Acknowledged Point To Point`, `Request` and `Response` are used to carry user data, i.e. set by the user when initiating its send request.

Public access feature

This is one of two features that are exclusive to the anisochronous mode. It is handled by the ACF sub-layer and uses the public access field within the frame's header to signal that the corresponding frame can be used by other nodes within the ring. The basic premise is that a node might not be operating at full capacity at all times. In other words, not all the node's three frames are continuously used for data transmission.

Two conditions must be satisfied before a frame can be reused, the frame's type is `Transport` and its public access field is set to 1. Thereafter, it can be used for both acknowledged and unacknowledged data transmission. In order to maintain ownership of the frame, certain header fields are treated as read-only by other nodes. Those fields are `OMAC`, `Frame identifier (ID)` and `Public Access (PA)`.

As soon as user activity resumes within the owner node, i.e. a new user send request is issued, the corresponding node automatically reclaims its frame(s). If the frame is not being used by another node, i.e. its type is `Transport`, then it is immediately reclaimed once it circles back to the owner. On the other hand, the owner updates the frame's public access field to 0, which signals that the frame can no longer be reused by other nodes. Once the corresponding data transmission is concluded, the frame can be reclaimed and used by its owner.

This feature significantly increases the network's performance at the expense of increasing the latency for some nodes within the ring. Nevertheless, the hard upper limit on latency is still maintained.

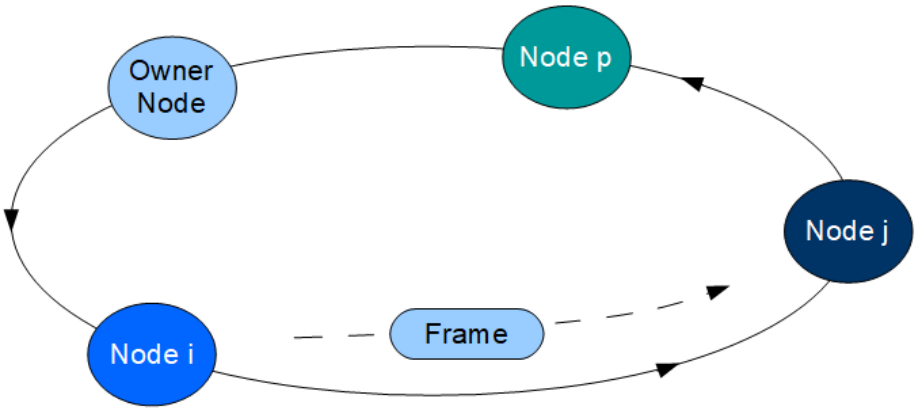


Figure 5-7: i-To-j feature illustration

i-To-j feature

This is the second exclusive feature of the anisochronous mode. Similar to public access, it is also handled by the ACF sub-layer. Unlike the previous feature, it neither requires the explicit permission of the frame's owner node, nor does it require a period of inactivity to reuse a frame. As illustrated in figure 5-7, a node i can reuse a frame to transmit data to node j, if the following conditions are satisfied.

- The frame's owner is not located between node i and j. Or if node j is also the owner of the frame. Using the frame's `OMAC` field, node i is able to deduce the owner's position, because MAC addresses are assigned incrementally within the ring.

- The frame's type is either `Transport`, `Positive` or `Negative Acknowledge`.

  - Two scenarios can result in a circulating transport frame. Either due to user inactivity within the owner node. Or due to a concluded unacknowledged data transmission, where the corresponding receiver node has updated the type field from `Ethernet Service` to `Transport` before sending the frame back into the ring.

  - As for the other two frame types, they resemble a returning acknowledge that is sent from the corresponding receiver back to the sender.

In case of a returning acknowledge, frame reuse is made possible due to the way acknowledges are tracked and processed within the ASA sub-layer. The only relevant information within a returning acknowledge is the frame's OMAC, ID and whether it is a positive or negative acknowledge. The rest of the header fields and payload can be modified including packet header.

Since both OMAC and ID are read-only fields, the EACK 2-bits field is used to embed the aforementioned positive or negative acknowledge. Thus, allowing frame reuse for a returning acknowledge. The first bit of the field indicates whether there is an embedded acknowledge within the header. And the second bit indicates whether the embedded acknowledge is positive or negative.

Due to the position-based condition, the frame reuse enabled by this feature is limited to unacknowledged data transmission. This can either be a user send request with `Ethernet Service` type. Or a `Acknowledged Point To Point`, `Request` and `Response` types with SACK set to 1 i.e. transmission of a routed packet. As described previously, such an acknowledged data transmission is treated as unacknowledged one beyond the sender's ring. More specifically, the i-to-j feature is disabled for a sender targeting a receiver within another ring. However, it is enabled for the router within the receiver's ring. Since acknowledges are skipped, such a router is able to use the feature not only for `Ethernet Service`, but also for `Acknowledged Point To Point`, `Request` and `Response`.

The aim of this feature is to keep the frame reuse transparent to the owner. Due to the aforementioned conditions and limitation, the performance increase due to this feature is less than public access. On the other hand, it does not increase the latency for any node within the ring and the hard upper limit on latency is intact.

Last but not least, frame reuse can be compounded because the two features can be engaged simultaneously. In other words, the i-to-j feature can be applied to an ongoing data transmission that was enabled due to the public access feature. In figure 5-7, consider that node p is using the frame due to the public access feature. Given that the above conditions are satisfied, node i can reuse the same frame, due to the i-to-j feature, to send data to node j. In such a scenario, SMAC is used instead of OMAC

to check for the frame's i-to-j feature eligibility.

SendAt feature

The main purpose of this feature is to extend CarRing IV support to time-triggered systems, i.e. not just event-triggered ones. It is handled by the ASA sub-layer and it can be used in both operation modes, isochronous and anisochronous. Typically, sending user data is done as soon as possible, i.e. with the next sending opportunity. In other words, as soon as the node's own frame circulates back and reaches the node's physical layer, then it is used to transmit user data. In case of the anisochronous mode, this also includes frames owned by other nodes.

This feature adds two time-specific data transmission options. The first option is to transmit user data at a future point in time. While the second option is to transmit the same user data periodically. In both options, the time value is provided by the user as part of the send request. Creating and initiating a send request as well as the user interface is covered in chapter 8. In short, the user send request must include three values.

- `SendAt`. It is a 2-bits value that indicates whether the corresponding user data must be sent immediately(value of 0), at a future point in time(1) or periodically(2).
- `Time interval`. This 40-bits value is used in combination with SendAt values 1 and 2.
- `State`. Using this 1-bit value, the user can either initiate(0) or abort(1) a data transmission.

As soon as the send request is received via the user interface, an internal counter is started using the time interval value, given that the State value is 0 and SendAt value is either 1 or 2. There are two internal counters, one associated with SendAt(1) and another with SendAt(2). Once the time interval elapses, then the send request is pushed by ASA to the ACF sub-layer for transmission. Thereafter, in case of a

delayed transmission (SendAt=1), the internal counter is reset. On the other hand, the internal counter is restarted in case of periodical transmission (SendAt=2).

The user is also able to abort a previously issued send request given that it was either a delayed or periodical one. This is accomplished by issuing another send-at request with the same SendAt value but a State value of 1. As long as the corresponding time interval has not elapsed, then the transmission is aborted. Otherwise, only the periodical transmission would be aborted. Note that the aforementioned send-at request does not require user data, i.e. a payload pushed with it.

In order to support delayed and periodical transmission, two additional memory allocations are used to store user data. Therefore, if multiple consecutive send-at requests were made prior to the time interval elapsing, then they would overwrite the previous send-at requests. As a result, the internal counters will be set based on the latest time interval and the corresponding user data in memory will be overwritten as well.

Recreating transport frames

The ACF sub-layer uses the artificial gap described above as part of a timeout-based method to recreate dropped frames by the physical layer. Since frames can be dropped by any node within the ring, the ACF sub-layer uses a continuously running internal counter that is based on the artificial gap. Once a frame is not received within the time limit, it is recreated and reintroduced similarly to the one used during ring initialization. In other words, the dropped frame is recreated and reintroduced based on the artificial gap and the frame's identifier. Therefore, the original frame succession within the ring and consequently the network's timing behavior is maintained.

5.2.7 Router vs Node design

Similar to the physical layer, the router design has two data-link layers. While the ASA sub-layers are an exact duplicate of the node design, the ACF sub-layers differ when processing a frame that contains a packet being routed. While the protocol

timing is maintained, there are additional scenarios considered within the sub-layer.

Within the receiver side of the ACF sub-layer, the frame is processed similar to a one destined for the router. In case of unacknowledged transmission, the frame's type is updated to `Transport` before being pushed back into the ring. However, in case of acknowledged transmission, then two scenarios are distinguished depending on the header field, `SACK`.

If the field is not set (i.e. 0), then a positive acknowledge is sent back by the router to the sender node. Since the corresponding packet is being routed and not received by the router, then ACF generates the positive acknowledge instead of ASA. As for negative acknowledges, they are only generated by ASA, if the packet is being received and `SACK` is unset.

On the other hand, if `SACK` is set(1), then it is treated similar to an unacknowledged transmission, i.e. converted to a `Transport` frame. In both unacknowledged and acknowledged transmissions, the frame is forwarded without any modifications, if the router is unable to route the packet.

As for the sender side of the ACF sub-layer, it includes an additional scenario that is part of network-wide clock synchronization. This is covered in detail in chapter 6.

5.3 Network layer

5.3.1 Overview

The data-link's sub-layers, features and addressing mechanism provides for data transmission within the same ring, i.e. local destinations. Not dissimilar to the OSI model [21], the main purpose of the network layer is to provide features and addressing mechanism that allows for across ring data transmission. It extends a CarRing IV network from a size of one ring with 16 nodes to 256 rings interconnected by routers. It supports both unicast and multicast routing. The latter is covered in chapter 7. The network layer includes measures taken to maintain a deterministic behavior and consequently a hard upper limit on latency. This includes the case of acknowledged

transmission and routed packets described previously.

In addition to the measures covered in sections below, the ability to prioritize user data is removed. More specifically, no priority header fields or priority queues are used. Other than the negative impact on latency, prioritization is considered as a patch or counter measure to possibly failing to match the real-time requirements of a system. In practice, there are systems where certain nodes have less demanding user applications than other nodes. In other words, a mix of user applications that have hard, firm and soft real-time requirements. The latter tolerates the occasional delayed transmission. However, the priority fix is rendered irrelevant as soon as all user applications have hard real-time requirements.

5.3.2 Services provided to application layer

Similar to the data-link layer, the network layer also operates in a connectionless-mode and it provides the following services:

- Packetizing. Same as a frame, packet has a fixed size and include a header, but no trailer. The encapsulation/decapsulation task is also delegated to the physical layer.
- Network address which essentially distinguishes rings from one another, i.e. a ring addresses.
- Unicast and multicast routing. The latter reuses the data-link multicast address and is covered in chapter 7.
- The network layer elevates some the functionality of data-link and makes it accessible to the application layer.
  - Acknowledged and unacknowledged data transmission.
  - Acknowledgment and destination found notifications. Note that such notifications do not apply for routed packets.
  - SendAt feature

5.3.3 Addressing and packet format

The network layer address is used to uniquely identify rings. It is 8-bits long and referred to as ring address. Unlike the OSI model [21], the addressing of this layer is not independent of the data-link addressing and does not uniquely identify a node. Rather, the combination of ring and MAC addresses form the node's address, which in turn uniquely identifies a node within a CarRing IV network. Therefore, there is no need for an auxiliary protocol that maps addresses between the two layers i.e. no Address Resolution Protocol (ARP)-like functionality is required.

The combined address is 12-bits in size and is represented using a dotted-decimal notation i.e (Ring address).(MAC address). The resulting address space is 2^{12} or 4096.

Unlike the node's MAC, the ring address is not dynamically generated during ring initialization, rather it is statically set as part of the primary node's configuration data. As described previously, it is assigned to nodes during initialization. Within the same ring, all nodes have the same ring address.

Figure 5-8 depicts the packet format. Since no fragmentation or segmentation is allowed, the network layer uses packets of fixed size, i.e. similar to data-link. A trailer is also not required because errors are handled by the physical layer. Unlike frames, a packet has a reduced header which includes only the following fields:

- **Source Ring Address (SRA)**. It is an 8-bits field that holds the ring address of the source node that is sending the user data.
- **Destination Ring Address (DRA)**. This 8-bits field is the counterpart of `SRA` and holds the destination ring address for the user data being transmitted.

There are no explicit type and retry header fields. Rather, they are handled internally by the network layer. The layer converts them into their respective frame header fields counterparts upon sending and vice versa upon receiving user data. While the retry field conversion is straightforward, not all frame types are available as packet types. More specifically, only the following packet types are available:

- `Ethernet Service`

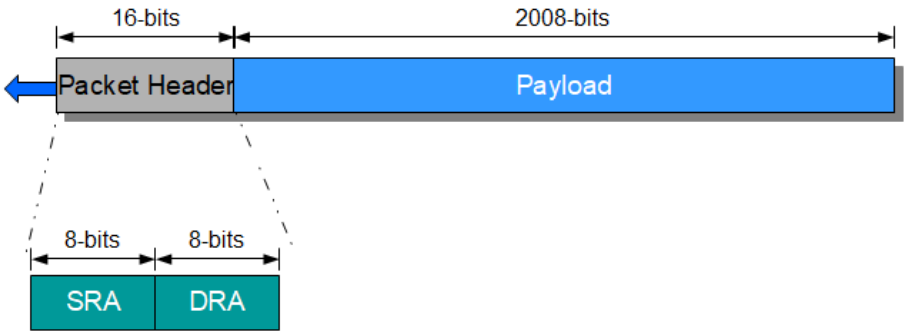


Figure 5-8: Packet format

- `Acknowledged Point To Point`
- `Request`
- `Response`
- `Multicast`

This internal conversion is transparent to the application layer. From the perspective of the application layer, all fields must be set and used when transmitting data, i.e. `SRA`, `DRA` as well as packet type and retry fields. Limiting the availability of certain frame types, effectively limits the user application control over the internal protocol processes. Otherwise, an unexpected `Reset` or `Acknowledge` would disrupt the internal data flow within the layers. Moreover, a smaller packet header reduces overhead and consequently improves the protocol's performance.

5.3.4 Local vs non-local destination

With the introduction of ring addresses, we identify two types of target nodes or destinations. When processing a user send request, these types are distinguished based on the sender's ring address and `DRA` field of the packet header. If they have the same value, which is the case when the sender and receiver nodes belong to the same ring, then the destination is said to be local, otherwise it is non-local.

The operation modes, described previously in the data-link layer, do not directly affect the behavior of the network layer. Rather, it is indirectly affected by whether

the destination is local or non-local. This distinction has implications in terms of the number of sending opportunities available for a packet. More specifically, one of the anisochronous mode features is disabled for non-local destinations.

The i-to-j feature is implemented for local destinations, since it only takes into account MAC addresses to enable frame reuse between nodes. Therefore, it can result in a false-positive, if used for a non-local destination. This is due to MAC addresses being unique only within the same ring, i.e. they constitute only a part of a node's address.

5.3.5 Static topology and routing protocol

Within a hard real-time system, the number of nodes remains constant. Any change to the system's internal components is performed offline. Thus, the number of applications, communication requirements and QoS constraints do not change during run-time. Furthermore, the senders, within the communication system, have a fixed number of destinations.

This static topology allows for the use of static routing. Due to the static nature of the network, the routing protocol does not require any exchange of information between nodes. Therefore, facilitating this exchange is not one of the roles of this routing protocol, neither is the maintenance of the router's convergence state. As a result, network traffic is entirely dedicated to user applications.

The information needed to route a packet is handled only by the router. This information is represented by a routing table that is calculated prior to network operation. Due to routers being able to connect only two rings, no explicit forwarding table is required, i.e. it is implicit. Furthermore, the possible destinations of each node in the network can be identified before deployment. Thus, routing paths do not change during run-time and consequently, computing paths is not done during run-time.

Since there is no exchange of information between routers and the network topology is static, the routing process functions are not part of the router functions. In other words, no auxiliary protocols required that update the routing table based on

changes in the underlying network topology. Thus, the router functions are simplified to include only forwarding functions. Those functions are limited to route look-up and packet forwarding. This effectively lowers the processing overhead within a router.

When a node sends data to a non-local destination, it does not need to know the address or number of routers within the ring. Rather, the router checks each circulating packet against the acceptance criteria as well as the routing table. Based on the outcome of the routing decision, packet forwarding is performed. The ability to examine each packet within the ring, stems from the way frames are handled on the data-link layer. In other words, as continuously circulating frames reach a router, their corresponding packets are examined.

5.3.6 Route computation algorithm and routing table

The protocol provides real-time guarantees for bandwidth, delay and jitter, which we define as our QoS. Prior to deployment, those QoS constraints are examined for each user application to check whether the network is capable of meeting them.

Within the same ring, user application requirements are checked against the ring's QoS attributes. Those vary based on the operation mode and size of the ring. However, for those user applications who have non-local destinations, a route must be selected. The route selection is based on a computation algorithm that takes into account the QoS constraints of the user application. Therefore, link costs are calculated in terms of QoS constraints.

In order to adapt to the multiple attributes of the QoS, two algorithms are considered depending on the dominant factor with respect to the user application. If delay and jitter are more dominant than bandwidth, then `Constrained Shortest Path First` is used. In other words, for delay and jitter the additive property is used to calculate the link cost between two nodes to determine the shortest path. As for the term Constrained, it refers to the initial step of the algorithm where the delay of each link that does not have adequate bandwidth for the user application is temporarily set as infinite, i.e. marked as unavailable. Therefore, it is a bandwidth constraint.

The second algorithm is used if bandwidth is more dominant than delay and jitter.

It is the `Constrained Widest Path First`, where the non-additive concave property is used to calculate link cost. Similarly, in the initial step, the respective residual bandwidth of links with inadequate delay is temporarily set to zero, i.e. marked as unavailable. Therefore, it is a delay constraint.

Regardless of the dominant factor, the least hop count variation of the corresponding compute algorithm is used. As for the shortest path first algorithm, the centralized approach [27] of Dijkstra is used, because there is no change in link cost over time.

As for routing tables, they are stored only in routers. A routing table is a fixed size look-up table of 256 entries, where each entry has a size of one bit, i.e. no need for Content-Addressable Memory (CAM). This fixed size is due to the ring address size, which is 8-bits. As for the one-bit entry size, it is due to router being able to connect only two rings, i.e. no port/interface specific information is required as part of the routing table or as part of a separate forwarding table.

A one-to-one mapping is established between all possible ring addresses and the routing table entries. Therefore, each entry corresponds to a ring address and the bit value stored within the entry resembles the routing decision. The bit value is interpreted as follows:

- **0:** Return packet to the same ring
- **1:** Route packet to the other ring

The bit value of each entry is set based on the result of the routing computation algorithm. And the default value is **'0'**.

5.3.7 Functional description

Unlike previous layers, the network layer functionality differs drastically between node and router. Therefore, this sub-section is split into two parts.

Node

In CarRing IV, a node does not store, pull or push any routing information. Therefore, the network layer functions are simplified.

Internally within the sender-side of the layer, processing send request is halted until the data-link layer signals that ring initialization is done. However, such a halt is not required in the receiver-side.

Once normal operation is started, the sender-side is responsible for creating the packet header as well as converting the type and retry header fields into their respective frame header fields. Similar to the data-link layer, the encapsulation and decapsulation is delegated to the physical layer.

On the other hand, the acceptance criteria is handled by receiver-side. After normal operation starts, there are frame types used when transmitting user data that do not have packet type counterparts. More specifically, `Positive` and `Negative acknowledges`. When receiving such frame types, the acceptance criteria of the network layer is ignored, i.e. they are handled entirely by the data-link layer.

As for local and non-local destinations, the only distinction is that the latter requires the sender-side to signal to the data-link layer that the i-to-j feature must be disabled for the corresponding send request. This is required to maintain separation of layers, i.e. ring addresses are outside the scope of the data-link layer.

Router

As described previously, the number of ports within the prototype is limited to two. As a result, a router routes and maintains packets between two rings. Therefore, it has two node addresses, each is associated with a router port.

As for the layer structure, each router port has its own network layer. However, those layers are not duplicates of the node's layer with relatively minor differences. Rather, only the sender-side of each network layer is a duplicate of the node's counterpart. In addition to router functions, the router's network layer also includes additional memory allocations that are separate and apart from those associated with

each port. Therefore, the router's network layer is considered as an own layer when compared to the node's layer, as previously depicted in figure 4-3.

As stated previously, a router has the same functionality as a node. This includes the ability to act as a primary within each ring it connects to via its router ports. Since a router has two addresses, it can be targeted simultaneously on both of its ports. In other words, a router is able to send and receive double the amount of user data of a node.

Since the basic sending and receiving procedures are the same as that of a node, the acceptance criteria within each port is limited to the address associated with that port. Within the same ring, if a sender targets a router using the address associated with its other port, then such a packet will not be received immediately upon reaching the router. Rather, it is routed to the other ring where it is sent and thereafter received again by the router itself. This is deliberate, because accepting a payload requires both data-link and network layers. In such a scenario, the performance is negatively impacted and the corresponding user application's QoS requirements might not be met. This scenario is attributed to an in-optimally configured user application.

Each router port has effectively two sources of user data or payloads, the application layer and routed packets. They are processed based on arrival time, i.e. first-come first-served. Theoretically, two payloads may arrive from the two sources at the same exact clock tick, thus competing for the same sending opportunity. In such a case, the selection method used maintains fairness. In the very first encounter of such an event, the routed payload is selected. However, the selection is memorized and used for the next encounter where the application layer payload will be selected. Essentially, alternating between the two sources, i.e. no priority.

As for routing functionality, it is part of the receiver-side of each network layer. As previously described, it has its own dedicated memory allocations, which is equal in size to the application layer's allocations. In other words, the two sources of payloads described above can issue the same number of send/route requests within each port. Within the context of queuing, the request queue associated with each source has the same length.

Each router port has its own routing table, i.e. a router has a total of two routing tables. Moreover, a router is implicitly capable of routing packets across rings that are directly connected to its ports. This is possible, because certain routing table entries are uninfluenced by the precomputed routes. In each router port, the routing table entry corresponding to the port's ring address is set to '0', while the entry corresponding to the ring address of the other port is set to '1'.

Unlike typical approaches, a sender does not send packets with non-local destinations to a specific router within the ring. In other words, a sender does not use, store or keep track of router addresses for non-local destinations, i.e. no gateway-like functionality. Rather, the packet is simply pushed into the ring. The first router encountered with an adequate routing table, routes the packet. However, if such a router is unable to route due to full queue, then the packet is pushed back into ring without any changes. While circulating and before reaching the sender, the next router with an adequate routing table and non-full queue, routes the packet. Otherwise, the packet circulates back to the sender, where a destination-not-found notification is issued to the corresponding user application.

This approach can be considered as a version of first-come first-served in a ring with multiple routers. Consequently, load balancing can be done by simply introducing another router in the ring without the need to update or modify the configuration data of any existing nodes or routers. Only the physical location of the new router within the ring must be considered, i.e. simple and very effective.

Within each port, routing is effectively disabled during ring initialization. Furthermore, each port monitors the initialization status of the other port within a router. In case of a router connecting two different sized rings, routing is disabled if the other ring has not yet finished its initialization. In such a case, packets with non-local destinations are processed similar to the case of full queue described above.

When routing a packet, the receiver-side of the port's network layer uses the data-link interface of the other port to issue a route request. From the perspective of the data-link layer, such a request is similar to that originating from the user application. Thus, no new frames are created for routed packets. As a result, they are not sent

immediately, rather they wait for a sending opportunity. This is crucial for the deterministic behavior and hard upper limit on latency, which would be otherwise affected by the increased number of circulating frames within the ring.

When creating a route request, the receiver-side of the network layer copies the packet's payload as well as part of the headers' information into its memory. While the `DRA`, `DMAC` and `SRA` fields are preserved, the `SMAC` field is overwritten by the corresponding port's MAC address. This is due to the routed packet being transmitted using frames circulating within the other ring. It is also done to enable MAC-based data-link features. Although sources and destinations are well-defined in a hard real-time system, if `SMAC` is still required then it is left to the user application to include it as part of the payload.

Last but not least, the aforementioned `SRA` field is additionally used as part of a limited counter measure to malformed routing tables. While a packet is traversing rings to reach its destination, intermediate routers additionally check the `SRA` field against the ring address of the other port. If they are equal, then the packet is not routed. To a certain extent, this measure can also stop a continuously routed/circulating packet within the network. Note that such a check is performed after routing table entry and queue status checks.

5.4 Application layer

5.4.1 Overview

Similar to other protocols, the application layer provides the means with which user applications can transmit data via a CarRing IV network. Unlike typical approaches, user applications within CarRing IV are not purely software-level applications. Furthermore, the application layer does not provide a software-style interface, i.e. API. Rather, it supports user RTL-level hardware designs and provides a hardware-level interface which consists of input and output ports as well as a timing diagram.

As described previously, the CarRing IV prototype includes additional transceivers

and interfaces. Those are separate and apart from the transceiver used to transmit protocol data. They have their own protocols, transmission mediums and networks. And they were introduced into the prototype for the application layer.

In addition to being available for user applications, the layer also has access to and makes use of those additional interfaces as part of its transparent mode. The latter is an application developed on top of the protocol. In short, it allows a CarRing IV network to transparently transfer data between user devices that are connected to those interfaces. Furthermore, it also allows CarRing IV to act as a backbone for other networks.

5.4.2 Services provided to user

The application layer operates in a connectionless-mode and provides the following services:

- Mechanism that handles and provides an interface for user request and feedback.
- Interface for receiving payloads.
- Access to the internally synced clock via a dedicated interface.
- Status and debug interface.
- Transparent mode which allows CarRing IV to act as a backbone for other networks as well as transparently transfer data between user devices.

5.4.3 Functional description

Rather than an API, the application layer provides four interfaces, each consists of multiple input and output ports. The implementation details, such as data-widths and timing diagrams, are covered in chapter 8. In short, the user abides by a timing diagram, which is provided alongside each interface. Such a diagram explains when and how to send and receive data using the interface.

This section covers the layer's functionality, however limited to send, receive and status interfaces. Whereas, the synced clock interface is covered in chapter 6. As for the layer's transparent mode, it is covered over multiple sections below.

When sending user data, a request must be created and pushed along with the corresponding payload through the request interface. Thereafter, the application layer follows-up with one or more responses via the feedback interface. Before delving into the various responses, the layer's sending behavior is covered.

Once a send request is received by the layer's sender-side, the user application can not interrupt or pause pushing the corresponding payload data. While such a feature can be supported, it is deliberately excluded because it would disrupt the internal timing of the protocol, which consequently affects the deterministic behavior as well as the hard upper limit on latency. Nevertheless, in case of interrupted payload data transfer, the missed or badly timed data are automatically replaced internally by zero-valued bits, i.e. the layer proceeds with the send request and the corresponding packet/frame is transmitted.

After the send request is received by the layer, the user receives either an `address -valid` or `-invalid` response via the feedback interface. The response contains an address validation result, the destination address as well as the corresponding frame's `ID` and `OMAC` associated with the request. Those values must be stored internally by the user application, while data transmission is not yet concluded.

With respect to the address validation result, the layer includes a limited destination address check, in-which the destination MAC address is checked in case of local destinations. This is possible because the ring size, i.e. node count is distributed during ring initialization. In such a scenario, the send request with an invalid destination address is still not interrupted, rather it is not processed further internally, i.e. no data transmission occurs. On the other hand, if the destination address is valid, then the address status response also indicates that the corresponding packet/frame is being transmitted. This limited address check is another measure to limit the effect of a malfunctioning user application within the network.

Depending on whether its an acknowledged or unacknowledged transmission, there

are one or more subsequent responses that range from `payload-routed` to `negative-acknowledge`. Those subsequent responses take place after the corresponding frame circulates back to the sender. In other words, they are not immediate like the address status response.

In case of unacknowledged transmission and a non-local destination, one subsequent payload status response is pushed to the user application. This is possible due to the way frames and packets are processed. If the corresponding frame circulates back to the sender unmodified, i.e. the same destination address and frame type, then the response is `payload-not-routed`. Otherwise, the counterpart response is pushed back to the user application via the feedback interface.

Similar to the address status response, the payload status response also includes the corresponding frame's `ID` and `OMAC`. Since send requests are processed in-order and based on arrival time, the user application can match the address and payload status responses using the `ID` and `OMAC` values.

As for acknowledged transmission, a retried transmission results in two payload status responses. Regardless whether the `Retry` bit is set, either a `positive` or `negative-acknowledge` response is pushed via the feedback interface. If negative acknowledge is received and the `Retry` bit is set, then another payload status response is pushed due to retransmission.

Regarding data reception, the user application receives a payload destined for its node by continuously monitoring the receive interface of the layer. The headers' fields are pushed followed by the payload data. Similar to the sending behavior, data reception can not be paused or interrupted. In acknowledged transmission, an interrupted reception does not affect the corresponding positive acknowledge. In other words, once headers' fields are being received by the user application, a positive acknowledge is simultaneously issued. Therefore, if the reception is interrupted there is neither a mechanism to change the already issued positive acknowledge to a negative one, nor is there a new negative acknowledge issued. Rather, this is considered as a malfunctioning user application.

Furthermore, the user application is given three clock ticks to detect and start

receiving the headers' fields and payload data. Otherwise, the application layer signals to lower layers that data reception is not possible. Thus, freeing the internal memory and the allocated resources of the processed frame/packet. This also results in a negative acknowledge in case of acknowledged transmission.

In other words, the payload might be partially or completely lost, but the network operation is not impacted by a user application failing to receive its payload. This loss of user data is attributed to a malfunctioning user application. Otherwise, the network operation is impacted, because processing of new incoming frames is blocked within the node or router.

Other than send and receive, the application layer also provides information about the current state of its node as well as the corresponding local ring i.e. not the entire network. This information is collected passively, i.e. it neither consumes nor requires additional processing time. Furthermore, the information is not fixed, i.e. it is not simply reflecting a fixed value from an internal memory allocation within the layer. Rather, it reflects the current values used by the protocol. Therefore, this information is also used for debugging purposes.

The status and debug interface is a read-only interface that provides the following outputs:

- **Ring state**. It is a 4-bits signal that reflects the current phase of ring initialization, namely `Assigning addresses`, `Distributing circulation period`, `Clock synchronization` and `Normal operation`.

- **Ring size**. This 4-bits signal is derived from the highest MAC address assigned during ring initialization.

- **Node address**. This 12-bits signal reflects the node's ring and MAC address.

- **Circulation period**. This 40-bits signal reflect the circulation period calculated during ring initialization.

- **Send count**. It is a 13-bits signal that reflects an internal send counter within

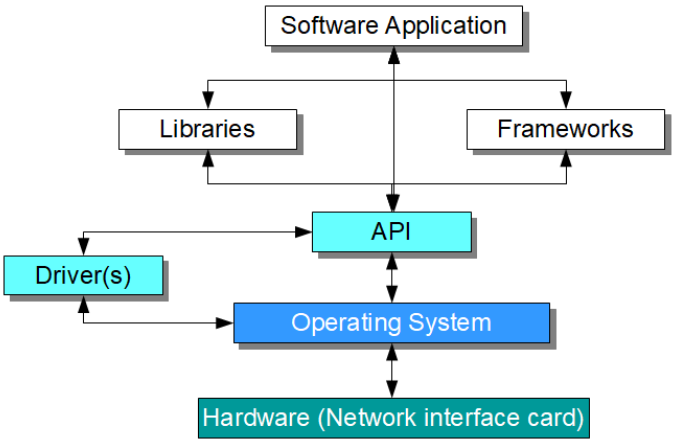


Figure 5-9: Typical components involved in software-level user application

the application layer. The counter is incremented only if the payload is transmitted i.e. failed send requests due to invalid addresses are not counted.

- **Receive count**. This is the counterpart to the above signal, and it is also 13-bits.

- **LayerX Activity**. This is a collection of 1-bit signals that reflect the activity of the sender and receiver side of each layer and sub-layer. A 1-bit signal is set, i.e. bit value is 1, when the corresponding layer or sub-layer has started to process a new frame/packet. And it is unset when processing is finished.

5.4.4 User applications

Typically, a user application is a software application developed using a programming language. It consists of one or more application processes that in turn use multiple other software components to transmit data. The main component is API, which is contained within the underlying OS.

The multiple components involved in such a software application are depicted in figure 5-9. Note the illustration in the figure is simplified, e.g. the distinction between filter and function drivers as well as user and kernel modes within the OS are beyond the scope of this thesis.

Typical network programming is done using a programming language like C or

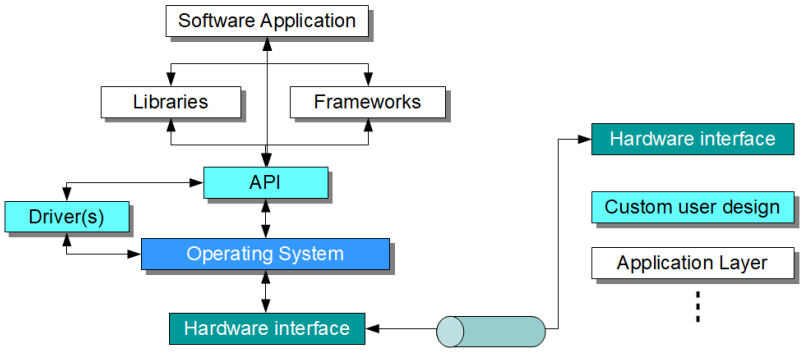


Figure 5-10: Second approach to creating a user application

Java. While this is also possible in CarRing IV, it is not the primary method with which a user creates applications for and interacts with the application layer of the protocol. As previously described in section 4.2, the usage of CarRing IV is broadly categorized into direct and indirect. In what follows, three approaches are covered, one is hardware-level only i.e. direct, while the other two are a mix of software and hardware-level user applications i.e. indirect.

Since all layers, including application, are implemented in hardware, the primary method is to create a custom RTL hardware design using an HDL. Such a custom user design consists of multiple hardware-level processes that in turn use the application layer interface to send and receive data.

The aforementioned approach of creating user applications is specific to RTL-level hardware designs. Software-level applications are also supported. Rather than listing all possibilities, two main ones are covered in this section.

As shown in figure 5-10, the second approach consists of a software application and a minimal wrapper-like custom user design. The latter is used as an intermediary between the application layer interface and the user's software application. More specifically, one of the supported physical interfaces within the prototype, e.g. Ethernet, is used for data exchange. When compared to hardware-only implementation, this approach requires minimal effort. The additional functionality required is limited to managing and creating pattern recognized by both the software application and minimal custom design. However, the performance might be limited by the used physical interface.

As for the third approach, the user simply re-purpose the transparent mode. In short, the transparent mode is a hardware-level application that allows CarRing IV to act as a backbone for other networks using the physical interfaces included in the prototype. It is covered in sub-section 5.4.6. Rather than a backbone incorporating multiple physical interfaces, the transparent mode is used for one interface by both the sender and receiver nodes. A wrapper-like minimal custom design is not required, since such a design is already part of the transparent mode design components. When compared to the second approach, this approach has one additional limitation. The software application is not able to dynamically set destination addresses. This is due to transparent mode original purpose, which is backbone-like functionality, i.e. other networks and interfaces do not dynamically change during run-time.

Note that in both software-level approaches, no new drivers, APIs, libraries or frameworks must be developed. Furthermore, no new patches or functionality is required or developed within the underlying OS.

5.4.5 Router vs Node design

Due to the router's two ports, data can be transmitted simultaneously on both ports. Therefore, the application layer includes duplicate request, feedback, receive, status and debug interfaces. However, it includes only one synced clock interface and has one transparent mode. The latter is directly affected by the number of supported physical interfaces within the prototype which remains fixed, i.e. the same prototype is used for node and router designs.

As described earlier, each router port has two sources of payload, routed packets and application layer. The resulting possible impact on performance is mitigated by the user's ability to transmit data simultaneously on both ports. Depending on the underlying network configuration, i.e. positioning of senders and receivers within the rings, this can effectively double the bandwidth available for the user application.

5.4.6 Transparent mode

This feature changes the application layer's normal mode of operation into what is referred to as the transparent mode. It eliminates the need for a software-level as well as hardware-level user applications. The user simply physically connects to one of the supported interfaces within the prototype board and begins transferring data via the CarRing IV network. Any data pushed from the user's end is automatically captured on the node end and transmitted to its destination. Thus, data is transferred transparently.

As for send requests, effectively the same request is continuously issued as new data is captured. The same headers' fields are used for every send request. Those are statically configured once within the node/router before run-time. In case of multiple destinations, multicast transmission is used.

Moreover, transparent mode is also capable of simultaneously transferring data between multiple supported interfaces. In other words, it is not limited to one supported interface per node/router. Thus, the combination of the transparent mode and multicast transmission allows a CarRing IV network to act as a backbone for other networks.

Despite having a physical connection to the prototype, the connection realized via the transparent mode is a logical one. Due to the protocol QoS and hard real-time guarantees, data from multiple supported interfaces can be transmitted without interference.

Transparent data transfer and backbone-like functionality allows CarRing IV to be easily integrated into existing systems. It can also be used to help transition an existing system's network into CarRing IV network.

In essence, transparent mode is a hardware-level user application developed on top of the application layer. As depicted in figure 5-11, the transparent mode consists of multiple design components most of which are external to the application layer. Only components that are responsible for issuing send requests as well as receiving incoming payload are encapsulated within the layer. A new send request is issued if

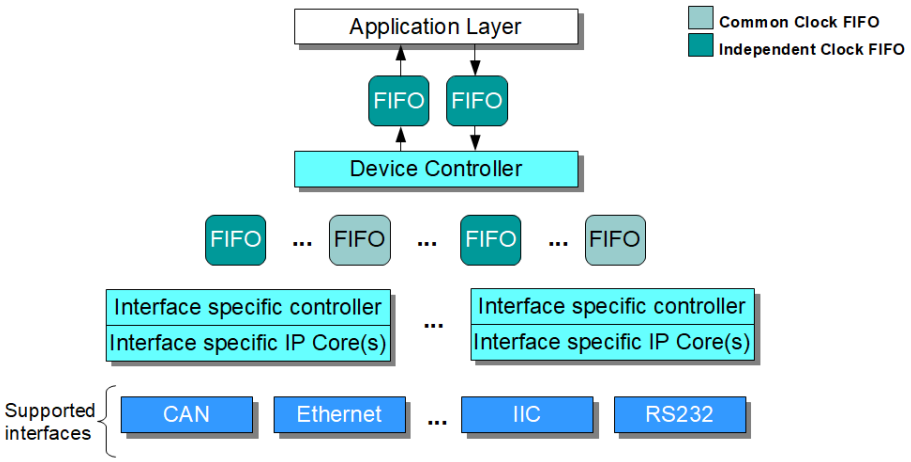


Figure 5-11: Illustration of the components that comprise the transparent mode

either the payload capacity is reached, or if there was a pause while retrieving new data from FIFO. As described previously, in the latter case, the payload is padded before transmission.

In what follows each set of components is described in a dedicated section.

5.4.7 Device controller

The main purpose of the device controller is to multiplex data from different interfaces. Unlike typical multiplexing approaches, the payload is not divided amongst the supported interfaces, where each interface has a dedicated portion of the payload. Rather, the payload is divided into fixed small portions. Those portions could all be carrying data from one interface or from multiple interfaces. This approach maximizes usage of sending opportunities, which simultaneously improves performance.

In order to identify which portion belongs to which interface, device controller frames are used. The corresponding frame format is depicted in figure 5-12. In what follows, a brief description of each header field is provided.

- **Interface identifier (ID)**. This 3-bits field is used to distinguish between the various supported interfaces.
- **Start Of Frame (SOF)**. Each interface has its own protocol and consequently, its own frame. The use of delimiters is crucial to handling different frame sizes

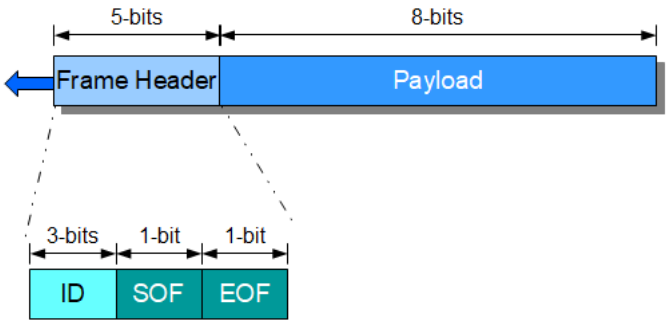


Figure 5-12: Device controller frame format

from different interfaces. This 1-bit field is used as a delimiter to identify the start of an interface frame.

- **End Of Frame (EOF)**. This 1-bit field is the counterpart of `SOF`.

Data exchange between the device controller and the application is split over two FIFOs, a device-to-application and an application-to-device FIFO. Similarly, two FIFOs are used to exchange data between the device controller and each interface.

When fetching interface data, the device controller continuously monitors all interface-to-device FIFOs. It then fetches data in a round-robin fashion from each interface. However, only one device controller frame worth of interface data per round-robin iteration is fetched. Otherwise, interfaces with relatively large frame sizes impose an unfair delay on other interfaces. To a certain extent, this approach is comparable to cell networking. In short, relatively small fixed-size data units, called cells, are used to multiplex different-sized frames.

5.4.8 IP Cores and auxiliary protocols

Each supported interface represents a separate network with its own protocols and layers. More specifically, each interface consists of chip(s) and a physical connector. In some interfaces, the chip is essentially a transceiver, i.e. only physical layer functionality. While in other interfaces, it additionally includes transceiver control logic or framing service. In other words, it includes a limited implementation of the first sub-layer of the data-link layer, i.e. MAC.

The transceiver of each interface is able to operate at different line rates or transmission speeds. Therefore, those transceivers must be properly configured before data can be transmitted via the interface.

In order to transmit data via an interface, an additional design component(s) is required. Such component(s) include control logic that utilizes the chip's internal registers to send or receive data. In short, the component includes a specific sequence of actions and register manipulations that must take place in order to operate the transceiver. A description of those sequence of actions as well as the chip's register map are provided by the manufacturer. Thus, each interface goes through a configuration phase, before data can be transferred.

For example, when using the CAN interface, one of the configuration steps is setting the proper bus speed. In order to achieve this, the chip's mode of operation must first be switched to the configuration mode. Thereafter, the proper bus speed is set by modifying the correct registers using the chip's register map. Once accomplished, the chip's mode of operation must be switched back to normal in order to transmit CAN messages.

As depicted in figure 5-11, each interface has its own custom controller as well as one or more IP cores. The interface-specific controller implements the control, configuration and transmission logic described above. As for the IP cores, they implement auxiliary protocols that are required to communicate with the chip. In other words, the chip has its own interface through which it can be accessed and controlled. Continuing the earlier example, SPI is used to configure the chip as well as push CAN messages for transmission.

Another very important consideration is the reference clocks used within the supported physical interfaces. Such reference clocks are used to drive the transceivers associated with each supported interface. Furthermore, each chip has its own interface, which in turn uses its own clock. This effectively transforms the transparent mode design from a single clock domain into multiple clock domain design.

In multiple clock domain designs, data or signals crosses from one clock domain to another. Such a clock domain crossing, introduces its own issues, which are but not

limited to meta-stability and re-synchronization. This is the main reason for using independent clock FIFOs between the device controller and the interface controller, as shown in figure 5-11.

5.4.9 Supported interfaces

As described previously, two prototypes were developed throughout the project's life-time, Virtex5- and Kintex7-based prototype. Although they include different physical interfaces, the same device controller frame format was used for both of them. As described previously, each interface has a custom controller and one or more IP cores. A functionality common to all interface controllers in both prototypes, is handling data-exchange via FIFOs between the interface controller and the device controller.

In what follows, the supported physical interfaces as well as their respective interface controller and IP cores are described.

Virtex5-prototype

Five physical interfaces are supported within this prototype. Three of which are transceiver-only interfaces, namely IIC, RS232 and Parallel port. While Ethernet and CAN, include the aforementioned limited MAC sub-layer of data-link.

In case of IIC, the interface controller is also responsible for encapsulation/decapsulation of FIFO data into IIC messages which are then used to transmit data using the underlying IP core. The IP core is a custom IIC core that operates as a master node and implements the IIC protocol. The implementation uses the 7-bit address space as opposed to the 10-bit extension. Furthermore, the core's frequency can be modified to operate in different speeds. Within the context of device controller frame, IIC interface `ID` is 3.

As for the IIC message, it is a 25-bits word used internally between the interface controller and IIC core. It consists of the following fields:

- **Identifier (ID)**. The IIC interface is considered as intermediary interface that is used to access and control another board. Two boards were developed alongside

the Virtex5-prototype. One controls the car headlight, while the other is for the car gas pedal. This 7-bits field is used to distinguish between such boards.

- **Access mode**. This 1-bit field identify whether the core should read from or write to the attached IIC device.
- **Device address**. This 7-bits field resembles the IIC device address.
- **Register address**. It is an 8-bits field resembles the register address within the IIC device.
- **Data**. This is the data byte associated with IIC transaction.

The parallel port is an 8-bits General-Purpose Input/Output (GPIO) internally connected to an 8-bits bidirectional voltage-level translator with auto-direction sensing. Therefore, only the custom controller is developed for this interface. Within the context of device controller frame, parallel port's `ID` is 5.

As for RS232, the interface controller also interacts with an Xilinx IP core which in turn implements UART. However, the IP core or XPS UART Lite is intended to be used as part of an embedded system. Therefore, it uses the Processor Local Bus (PLB) interface. Thus, the interface controller additionally implements the PLB interface and operates as a PLB master. Within the context of device controller frame, RS232 interface `ID` is 4.

Unlike RS232, the CAN interface chip does not use a PLB interface, rather it uses SPI. Thus, the interface controller also interacts with an Xilinx IP core that implements SPI. Similar to RS232, the IP core or XPS SPI is also intended for embedded systems. Thus, the interface controller also implements PLB and operates as a PLB master. However, the implementation is more complex than RS232. It implements two sets of action sequences and register manipulation. The first controls SPI, which in turn controls interface's chip. Within the context of device controller frame, CAN interface `ID` is 1.

With respect to Ethernet, the interface controller neither interacts with an Xilinx nor custom IP cores that implement the protocol. Rather, it interacts with an Xilinx

Ethernet MAC Wrapper which in turn uses the LL interface, i.e. similar to Xilinx Aurora. Thus, the interface controller additionally implements the LL interface.

The used Virtex-5 FPGA already includes an embedded Tri-Mode Ethernet MAC, which can operate at 10 Mbps, 100 Mbps and 1 Gbps. Thus, the Ethernet interface must be properly configured with the adequate mode/speed before data transmission. The corresponding chip uses SMI. Therefore, an additional custom IP core is developed which implements SMI. Within the context of device controller frame, Ethernet interface `ID` is 2.

Kintex7-prototype

Unlike the previous prototype, the Kintex7-prototype does not directly include physical interfaces. Rather, it includes intermediary interfaces that connect to readily available third-party boards. In turn, those boards include the physical interfaces. Three intermediary interfaces are included within the prototype, namely RaspberryPi, MikroBUS and Pmod. Therefore, the interface identifier field size within the device controller frame is still adequate for this prototype.

Each intermediary interface includes multiple auxiliary interfaces/protocols. In case of RaspberryPi interface, it includes two SPI, one UART and one IIC interface. As for the MikroBUS interface, it includes one SPI, one UART and one IIC interface.

On the other hand, the Pmod interface does not include explicit auxiliary interfaces. Rather, it has multiple I/O assignments for its interface pins. Each I/O assignment includes one auxiliary interface/protocol. The Pmod interface version used in the Kintex7-prototype consists of 12 pins as opposed to 6. Thus, the interface supports GPIO, SPI, UART and IIC as well as their respective expanded version. In short, the expanded version uses all 12 pins instead of 6, where most of the additional pins are used as GPIO.

Similar to the approach used in the previous prototype, Xilinx IP cores are used that implement the protocols associated with each auxiliary interface. Likewise, those cores are intended for use within embedded systems. However, instead of PLB, the cores use Advanced eXtensible Interface (AXI), which is part of Advanced Microcon-

troller Bus Architecture (AMBA). More specifically, they use AXI4-Lite.

Chapter 6

Clock Synchronization

This chapter explains the network-wide clock synchronization approach used in CarRing IV. While synchronization takes place during ring initialization, the design component(s) involved are used by and exist outside of protocol layers. Therefore, the approach requires its own dedicated chapter, which is introduced in the first section. Then, the timer module and its operation is described. Thereafter, synchronization within a single ring as well as across multiple rings is covered.

6.1 Introduction

Within a real-time system, communications can be considered as either event-triggered or time-triggered. In addition to event-triggered, CarRing IV also includes support for time-triggered communication via its `SendAt` feature. In short, it allows for periodical data transmission as well as at a future point in time. The corresponding time values are based on the node's local clock/counter. As described in chapter 2, clocks here refer to hardware-level signals as oppose to software-level clocks or clocks within the context of embedded systems, i.e. actual time in hours-minutes-seconds.

The main purpose of clock synchronization is to further extend CarRing IV's support for time-triggered communications. More specifically, it synchronizes all nodes within the entire network to one reference node. Since synchronization takes place during ring initialization, only a primary node can be used as a reference node.

Unlike typical ping-pong style, the approach used does not require exchange of messages between each node within the network and the reference node. Furthermore, there are no dedicated packets that traverse the entire network to achieve synchronization, i.e. no routing required. Rather, nodes are synchronized either if they are in the same ring as the reference node or if one of the routers within the same ring is synchronized. In other words, the reference node need not be aware of all nodes within the entire network, rather only those within its own ring. Thus, the approach is scalable.

The end result of this synchronization is syncing an internal design component that exists within each node and router, called timer module. It is used by all layers for any timer- or time-out related functionality, e.g. acknowledge time-out, reliable multicast time-out, transport frame recreation, `SendAt` periodical and delayed transmission, calculating circulation period as well as distributing it to all layers.

6.2 Timer module

Similar to real-time embedded systems or more broadly software-level solutions, timers are an essential part of managing time-sensitive events, e.g. timers are used in scheduling events or managing time-outs. Such timers can be classified [23] as software-level i.e soft timers, or physical timer chips i.e. hard timers. Although CarRing IV includes a hard timer, it is based on clock cycles or ticks rather than time-stamps or hours-minutes-seconds clocks. Nevertheless, the time values used in the protocol can also be expressed in seconds by simply multiplying the number of clock cycles elapsed by the period of the clock that drives the protocol design, e.g. 100 cycles x 6.4 ns = 640 ns.

CarRing IV timer is a separate design component that exists outside the protocol layers. In essence, it is based on a free running counter that increments (+1) with each clock cycle. Consequently, the protocol's time resolution is equal to the design-clock's period. Thus, all time values provided by the user to or retrieved from the protocol are expressed in number of clocks rather than seconds.

The timer consists of multiple clocked processes and inter-process communication channels. Those processes handle the circulation period, synchronizing the main counter within the timer and synchronization-related operations. The latter includes calculating the processing time per link/node as well as handling the corresponding unsigned integer "division" operations. Those operations include non-multiple of two divisions which are based on shift-right and add operations.

As described in sub-section 5.2.6, a ring must be initialized before data transmission can take place. During the first phase of the initialization, the circulation period is calculated. It represents the amount of time that the `Reset` frame requires to circulate a ring. Within a ring, only the primary node calculates and distributes the circulation period to other nodes, i.e. secondaries. Thus, the circulation period requires its own clocked process within the timer.

In order to manage the start and stop conditions for the circulation period counter, this clocked process interacts with the ACF sub-layer of the data-link layer. The counter for the circulation period starts as soon as the ACF sub-layer signals that the `Reset` frame transmission has started. On the other hand, the counter stops as soon as the ACF sub-layer signals that the `Reset` frame is being received.

The process also handles the possibility of a dropped `Reset` frame. This might occur within the physical layer of the primary node or any other secondary within the ring. Therefore, a time-out based approach is used to detect such a dropped frame. The time-out value is statically configured within each node and router before runtime. The value is based on an estimate that is calculated using a simulated version of the ring. Thus, each ring size has its own time-out value. The simulation also takes into account the timing behavior of the physical layer. Once a dropped `Reset` frame is detected, the timer signals a time-out to the ACF sub-layer, which in turn creates and sends another `Reset` frame.

In order to sync the timers within each node, the main counter within each timer must be adjusted to match the reference node's main counter. This adjustment is based in part on calculating the processing time per link/node. Within each node, such calculation is performed during the second phase of the ring initialization, i.e. as

soon as the circulation period and ring size values are received. As shown below, the calculation performed within each node involves the ring size, circulation period as well as the `Reset` frame processing time within a node. The latter is fixed value that is derived from the protocol design. While having a fixed value, data transmission time depends on the underlying Aurora core configuration and customization. In other words, it depends on the physical layer's properties.

$$TotalProcessingTime = (RingSize - 1) \times ResetFrameProcessingTime \quad (6.1)$$

$$PerLinkTime = \frac{CirculationPeriod - TotalProcessingTime}{RingSize} \quad (6.2)$$

The division used in equation 6.2 is based on shift-right and add operations. And it is delegated to multiple other clocked processes. Each process is responsible for one of the ring-size possible values. Only non-multiple of two values have their dedicated processes. Thus, equations 6.1 and 6.2 have their own process which in turn communicates with a division-process based on the corresponding ring size.

So far, all the processes described do not handle the timer's main counter. Rather, the counter has its own dedicated clocked process. In addition to the +1 increments, the process also covers two possible scenarios that can take place during the clock synchronization phase of ring initialization. An additional frame header field, called `State`, is introduced and used to keep track of and handle those scenarios across all nodes within the ring. As previously described in sub-section 5.2.6, this field only exists within the context of a `Reset` frame, i.e. if the frame type is `Reset`. It is a 2-bits field that is used in conjunction with the `Phase` field of the `Reset` frame header. As for the two scenarios and the `State` field values, they are described below in two dedicated sections.

6.3 Synchronization within one ring

Synchronization within one ring accounts for one of the scenarios mentioned above. In this scenario, the ring being synchronized contains the reference node. As the circulation period phase is concluded, the primary node updates the `Phase` and `State` fields of the `Reset` frame header to start the third and final phase of ring initialization i.e. clock synchronization. Since in this case the primary is also the reference node, it sets the `State` field value to 2. Furthermore, the primary includes the value of its main counter within the frame's payload. This value is referred to as Sync value.

A `State` value of 2 indicates to all secondaries that they must sync their main counters using the sync value included in the frame's payload. Within a secondary's timer module, the clocked process that handles the main counter interacts directly with the physical layer to retrieve the aforementioned value from the payload. The sync value can not be used directly to sync a secondary's main counter. As shown below, two offsets must be included in order to accurately sync the main counter.

$$\begin{aligned} TransOffset = (OMAC - 1) \times ResetFrameProcessingTime \\ + OMAC \times PerLinkTime \end{aligned} \tag{6.3}$$

$$MainCounter = SyncValue + TransOffset + InternalOffset \tag{6.4}$$

The transmission offset or TransOffset calculates the amount of time elapsed before the `Reset` frame reached the corresponding secondary node. Since MAC addresses are assigned incrementally within the ring, the offset can be calculated using the secondary's own MAC, which also acts as an indicator of a node's position within the ring. Thus, the transmission offset is the summation of accumulated processing time within predecessor nodes, i.e. (OMAC - 1) nodes, and the accumulated time within links traversed, i.e. OMAC links. As for the internal offset, it accounts for the additional time consumed within the main counter's process, i.e. internal processing time which includes interacting with the physical layer.

6.4 Synchronization across rings

The second scenario involves synchronizing rings that do not contain the reference node. In this case, the corresponding primary node must first search for a secondary with a synced timer module and retrieve the corresponding sync value from that secondary. Thereafter, it must use the retrieved sync value to synchronize itself and all other nodes within the ring. Due to the router's two ports, a router's timer module could be synced due to a completed ring initialization in one of its two rings. Thus, the aforementioned synced secondary can only be a router.

Similar to the previous scenario, the primary node updates the `Phase` and `State` fields and starts the final phase of ring initialization. However, it sets the `State` value to 0 which indicates that the primary is looking for a synced secondary. The ring remains in the clock synchronization phase of the initialization until at least one of the secondaries/routers is synced. As soon as the `Reset` frame reaches a synced secondary, the router updates the `State` value to 1 and sets the header's `SMAC` to its own MAC address. The router also includes the value of its main counter in the frame's payload. In case of multiple synced secondaries, the aforementioned header fields and sync value are overwritten by the most recent router encountered.

Once the `Reset` frame reaches the primary node, the header's `SMAC` as well as the corresponding sync value are used to sync the primary's main counter. As shown below, the primary's timer module uses similar equations to those described in the previous section.

$$\begin{aligned} TransOffset = (RingSize - SMAC - 1) \times ResetFrameProcessingTime \\ +(RingSize - SMAC) \times PerLinkTime \end{aligned} \tag{6.5}$$

$$MainCounter = SyncValue + TransOffset + InternalOffset \tag{6.6}$$

After the primary node is synced, it updates the corresponding `Reset` frame and proceeds to sync all other nodes within its ring. This synchronization is done in the

same manner as described in the previous section.

The approach used to synchronize all rings within the network does not require routing, i.e. no need for dedicated packet that traverses rings to achieve synchronization. Furthermore, the reference node does not need to be aware of all nodes within all rings, rather it is only responsible for synchronizing nodes within its own ring. Thus, the approach is scalable.

The approach is made possible due to the way transceivers are configured in the router's physical layers. While the transceiver-related implementation specifics are beyond the scope of this thesis, in what follows a brief description is provided.

Each router port has its own physical layer, i.e. its own Xilinx Aurora core. An Aurora core relies in-part on a reference clock, which is a differential clock signal. The aforementioned design-clock is an output of the Aurora core. In a router design, the two Aurora cores associated with the two ports are driven by the same reference aurora-clock. And consequently, the same design-clock drives both ports of the router. As a result, both router ports are in same clock domain which means that there is no need for crossing clock domains. Within the context of the protocol, this implies that one timer module is sufficient for both router ports, i.e. if one port is synced then the other port is also considered synced.

Furthermore, the Aurora cores within both node and router designs are configured using the same basic parameters. In other words, the design-clock signals that drive router and node designs have the same frequency. Thus, the sync values included within a frame's payload can be directly used, since they represent the same value in seconds.

Last but not least, the size of a ring directly affects the amount of time required to finish its initialization. Therefore, in a multi-ring network, some rings might finish initialization before others. In such a scenario, data transmission is not halted in the entire network, rather only in those rings that are still being initialized. Moreover, packets can not be routed through to such rings regardless of the corresponding routing table entries. Such a routing decision affects both unicast and multicast transmission.

Chapter 7

Reliable Multicast

This chapter covers CarRing IV reliable multicast as well as multicast routing. Since the service spans multiple layers within the protocol, it requires its own dedicated chapter. The first section introduces the service and its background. Then, the service's description and functionality is divided into two sections based on the layers involved, i.e. data-link and network layers.

7.1 Introduction

Typically, multicast service is used for applications such as streaming media, teleconferencing or distributed databases. While infotainment systems are part of in-vehicle communication, i.e. one of CarRing IV application areas, the service is introduced and implemented for two main reasons, emulation of field buses and extending transparent mode functionality.

A primary example is CAN bus. Using multicast and one of the prototype physical interfaces, CarRing IV is able to support bus communication between multiple nodes/routers. As for transparent mode, it is no longer limited to one-to-one communication provided by unicasting. Using multicast, it can also provide a backbone-like functionality in-which a CarRing IV network can additionally support field buses via the prototype's physical interfaces.

The multicast service implements the one-to-many model, i.e. no support for

many-to-many or many-to-one. Furthermore, its implementation reflects genuine multicast as opposed to multiple unicasting, which negatively impacts latency. It uses a tree-based hierarchy, which bounds the sender's processing time by the number of its immediate receivers within the same ring.

Additionally, the service also implements reliability that is neither strictly sender-initiated nor receiver-initiated and it does not suffer from acknowledgment implosion. Since reliable multicast implicitly involves retransmission of packets, such retransmission might not be desired by certain applications, e.g. the payload contains a command that must be executed within a certain time limit, or a greater importance is placed on receiving a continuous stream of data rather than receiving every payload. Thus, reliability is introduced as an optional feature that can be enabled or disabled with every multicast transmission.

Last but not least, broadcast is not implemented by the protocol. It constitutes a special case of multicast where all network nodes are members of the same group. This design decision also avoids performance issues that are caused by misuse of a broadcast delivery service.

7.2 Data link layer

Multicast is implemented at the data-link layer, which allows for near wire-speed and reduced latency as well as reduced protocol complexity, when the underlying network is a single ring.

With respect to the data-link sub-layers, ACF handles the acceptance criteria as well as data transmission. Whereas, reliability and retransmission are handled by the ASA sub-layer. In order to support retransmission, ASA also has two additional memory allocations, i.e. a maximum of two concurrent multicast transmissions where reliability is explicitly enabled.

As for data-link operation modes, the i-to-j feature is disabled for multicast transmission. In essence, the feature enables frame reuse if the re-usage can be concluded before the frame reaches its owner node or original destination. By definition, a

multicast frame must circulate the entire ring.

Only the anisochronous mode's other feature, i.e. public access, can be used for multicast. This is possible because public access temporarily hands over the control over the frame from its owner node to other nodes. Furthermore, if the frame is still being reused, the owner node must wait for it to circulate the ring at least once before it can be reclaimed.

7.2.1 Addressing and frame format

Unlike typical approaches, the multicast service has its own addresses that are separate and apart from data-link and network layer addresses. Furthermore, there is no data-link and network layer variations of the multicast address as is the case for the TCP/IP protocol suite. Rather, one type of multicast address that is used and processed by both layers.

The multicast address is 8-bits in size and presented as a decimal digit. As a result, up to 256 multicast groups can be created. It is unique network-wide. Thus, it can also be used by the network layer.

Due to the reliability feature, multicast extends the frame's header beyond other frame types, from 24-bits to 40-bits, while keeping the overall frame size fixed. This is made possible due to the dynamic interpretation of the headers' structure. As previously described, the internal structure of headers differs based on the frame's type which is the very first field processed in an incoming frame.

In order to accommodate this extension, the packet header is completely removed. However, this does not affect the network layer functionality with respect to multicast transmission. Figure 7-1 depicts the format of a multicast frame. As for the frame header, certain fields are preserved while others are removed and replaced by new ones.

The `Type`, `OMAC`, `ID` and `SMAC` fields are crucial for basic functionality and are therefore preserved. Since the public access feature can be used for multicast transmission, the `PA` field is also preserved. Below, a brief description of each newly introduced header field is provided.

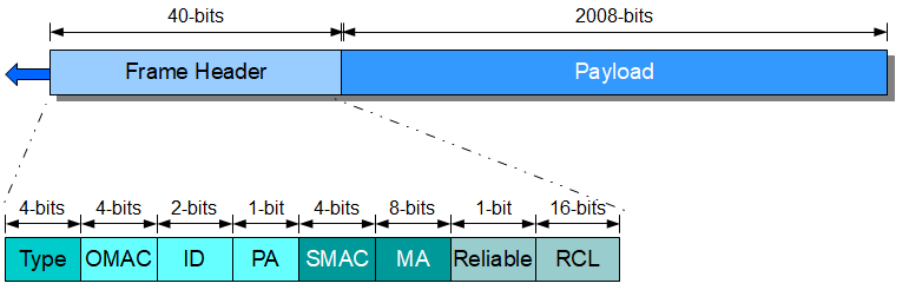


Figure 7-1: Multicast Frame format

- **Multicast Address (MA)**. It is an 8-bits field that holds the multicast address, which is used by both the sender and receiver nodes.

- **Reliable**. When set, this 1-bit field indicates that a retransmission must be attempted, if not all multicast group members have received the payload within the ring.

- **Receiver Check List (RCL)**. This 16-bits field is used as part of the reliability feature, which is covered in a dedicated section below. In short, as the multicast frame circulates, each multicast group member updates one bit in the field upon receiving the corresponding payload. Based on bit order and MAC address, each bit corresponds to a node within the ring.

7.2.2 Group membership

CarRing IV protocol targets hard real-time systems which are well-defined. Therefore, group membership is managed statically, which eliminates the overhead associated with monitoring the group, processing join/leave requests as well as sending group status updates. As a result, network resources are dedicated to transmitting user data.

Each node is equipped with a multicast group membership table. The membership table can be considered as a two-dimensional look-up table. There are a fixed number of rows, which is based on the multicast address size, i.e. 256 rows. However, the number of columns depends on the ring size. Each cell is 1-bit in size and represents

one of the nodes within the ring. Thus, the overall size of the table depends on the ring size, i.e. up to 16 columns max which reflects the max ring size of 16 nodes.

A node's group membership is indicated by the bit-value of the corresponding cell, i.e. '1' for member and '0' is not a member. All cells have a default bit-value of '0'. For example, in order to check a node's membership of a group, the corresponding multicast address is used to select the row. Thereafter, the node's MAC address is used to select the column. And the membership is indicated by the bit-value of that cell.

Last but not least, the membership table is not specific to a single node. In other words, it also includes the memberships of all nodes within the ring. Thus, the same table is statically configured within all the nodes, regardless whether the node is a member of any group.

7.2.3 Reliability

Typically, reliability in multicast is achieved using acknowledgments. Since the communication model is one-to-many, acknowledgments introduce a major problem referred to as acknowledgment implosion. In short, the problem stems from the requirement that each successful reception is followed by a unicast acknowledge sent from the receiver to the sender, i.e. the bigger the multicast group, the more acknowledges the sender must process. Negative Acknowledgments (NAKs) were introduced to combat this issue. It is sent by the receiver only if an error or a missing multicast packet is detected. Such an acknowledge is sent either as a unicast to the sender or as a multicast to the group.

According to [22], approaches to reliable multicast can be classified as either sender-initiated, receiver-initiated, receiver-initiated with NAK-Avoidance, tree-based or ring-based. Depending on the approach used to implement multicast reliability, acknowledgments or negative-acknowledgments might impact network performance.

Reliability in CarRing IV multicast does not require explicit acknowledgments or negative acknowledgments. Therefore, it does not suffer from the problems associated with them. As a result, schemes like NAK-avoidance are not required, which in turn

reduces the protocol's complexity.

As the multicast frame circulates the ring back to sender, each group member updates the value of only one bit within the RCL header field. Within the field, bits' positions correspond to nodes' MAC addresses. Thus, using the node's MAC address, the corresponding bit is set to '1' for payload received and '0' if it was not received. With each transmission of a new multicast frame, all bits within the field are set to '0'.

Unlike the protocol's acknowledged transmissions, acknowledges are embedded into the multicast frame. And consequently, one frame is sufficient to multicast and confirm the successful reception of a multicast payload. Thus, CarRing IV protocol uses the sender-initiated approach within a single ring.

7.2.4 Functional description

Since the header contains fewer fields, the corresponding multicast send request also requires fewer fields, namely `MA` and `Reliable`. If the `Reliable` field is unset(0), then the corresponding internal memory and allocated resources associated with the send request are freed as soon as the frame is sent. Otherwise, they are kept until the transmission is concluded, i.e. successful reception or retransmission has already been attempted.

In case of retransmission, the sender uses the `OMAC`, `ID` and `MA` fields to correctly identify which multicast payload must be retransmitted. When the multicast frame returns to the sender, the `RCL` field is compared against the corresponding membership table entry. Rather than selecting cells, all columns of the corresponding row are used as a single field and compared against a trimmed version of the `RCL` field. Trimming is required, because the number of columns is based on ring size i.e. not a fixed 16. `RCL` field is trimmed using the ring size value, which is distributed during ring initialization. With the exception of the senders own bit within both fields, retransmission is required, if they are not equal. Also the `RCL` field of the returning frame is reused again without modifications during retransmission.

Similar to acknowledged transmission, retransmission can also be triggered due to

a time-out. If the multicast frame does not return to the sender within a certain time limit, then retransmission is required. The time limit is also based on the circulation period.

Unlike acknowledged transmission, retransmission of a multicast frame is immediate, i.e. using the same original frame that returned to the sender. In other words, it does not require creating another send request and consequently waiting for another send opportunity. This is due to i-to-j feature being disabled for multicast. During transmission, a multicast frame is not converted into another frame type by a receiver, which in turn might be reused via the i-to-j feature.

Furthermore, acknowledgments are embedded in case of multicast. Consequently, there is no explicit acknowledge-frame that can be reused via i-to-j. In other words, CarRing IV's reliable multicast is not an extension of its acknowledged transmission. This is why there is an explicit distinction between the `Retry` and `Reliable` fields both in protocol design and implementation.

Unlike unicast transmission, accepting a multicast payload is not a combined decision of data-link and network layers. Rather, it is done in data-link based on the `MA` field and membership table. Thus, the membership table is used while receiving a multicast frame as well as retransmission. Within the same ring, `SMAC` is used to identify the sender node. As for the network layer, it is only concerned with routing.

Due to retransmission, a group member might receive the same payload twice. In order to prevent such a scenario, an additional check is performed before the payload is pushed all the way to the application layer. A receiver node simply checks the value of the bit that represents its own MAC address within the `RCL` field. If the bit value is '1', then the corresponding payload was already received and is therefore, not pushed to the application layer. Otherwise, the bit value is updated to '1' and the application layer gets the payload.

In addition to the feedback associated with sending and receiving a multicast frame, the data-link layer of a receiver also includes the `RCL` field as part of its feedback to higher layers. Thus, each group member can also provide the reception status of other members as well. Due to the physical ring topology, it is only limited to members

that precede the node itself within the ring.

7.3 Network layer

The network layer is only concerned with multicast routing. Therefore, the layer's implementation of multicast can be considered as an extension, which allows multicast traffic to be routed across rings.

This extension does not require address translation or mapping between network and data-link layer addresses, e.g. address translation from class D IP addresses to Ethernet MAC addresses, where the resulting translated address is not unique. Instead, the network layer directly uses the same multicast address introduced within data-link.

Unlike acknowledged transmission, retransmission is also supported for routed multicast packets, i.e. in other rings beyond the sender's own ring. However, retransmission is not performed by the original sender for the entire network nor is it performed by a core or designated router, i.e. unlike typical approaches. The design decision to supported retransmission is due to the impact of the multicast transmission, i.e. it involves multiple receivers rather than one.

7.3.1 Packet format

As previously described, the packet header is completely removed in order to implement the reliability feature. This is possible, because a multicast address is unique network-wide. Furthermore, ring addresses are not required for multicast routing. Nevertheless, the header is still used internally within the protocol layers. A multicast packet header consists of packet type, `MA` and `Reliable` fields. Similar to unicasting, the fields are converted internally into their respective frame header fields counterparts.

7.3.2 Reliability

As described previously, the sender-initiated approach is used within a single ring. However, across multiple rings, the approach can be best described as tree-based. In essence, the tree-based approach divides the multicast group into smaller groups each with their own group leader that is responsible for retransmission.

With respect to CarRing IV multicast, each ring within the network is treated as a sub-group. The sender node is the group leader within its own ring. As for other rings, the router that is responsible for routing the multicast packet is the respective group leader for the destination ring.

Whether be it the original sender or a router, each group leader is only aware of the immediate receivers within its ring and not all the receivers across the entire network. Consequently, ensuring a reliable transmission for a sub-group is the responsibility of the group leader and not the original sender. Therefore, this approach is scalable and can be seen as a mix of sender-initiated and tree-based.

7.3.3 Multicast routing

CarRing IV multicast routing can be considered as the multicast extension of its unicast routing. It also uses static routing and it does not require any exchange of information between nodes or routers. Similarly, the network traffic is also dedicated to user applications.

Multicast routing uses the source-based tree approach. This approach is efficient for the protocol since the number of possible multicast groups is limited to 256, i.e. it is not in the thousands. As opposed to group-shared tree approach which is used, if there is a large number of multicast groups. Due to the use of a core router, the group-shared approach suffers from a single point of failure.

Similar to unicast routing, the route computation algorithms are also run offline to build a separate tree for each source. The resulting multicast routing table is statically configured within each router before run-time. With respect to the algorithms, the multicast extensions of those previously described in unicast routing are used.

As for multicast routing tables, they have the same structure and size as the unicast routing tables. However, the 256 entries are based on the multicast address, which is also 8-bits. The bit value of each entry is interpreted similarly, i.e. '1' for routing and '0' for returning the packet to the same ring. The default value is '0'.

7.3.4 Functional description

Since the network layer is only concerned with multicast routing, its functionality within a node is limited. It handles multicast send requests coming from the application layer in which it converts packet header fields into their respective frame counterparts. Much like `Positive` and `Negative` acknowledges, the acceptance criteria is entirely handled by the data-link layer.

Multicast routing is done in the same manner as unicast routing. In addition to having its own membership table, each router port has its own multicast routing table that is used to make the routing decision based on the multicast address. However, there are two additional considerations, handling the `RCL` field and post-processing of a multicast packet.

To better understand the implications of the router's dual functionality on the `RCL` field, consider the case of a router that is routing a multicast packet, while simultaneously being a member of the corresponding multicast group. The router port's bit within the `RCL` field can convey only one piece of information. Thus, it can not be used to avoid both duplicate reception and routing in case of retransmission. Rather, those are monitored internally within each router port.

Within the `RCL` field, the router port's bit is set if the multicast packet can be received and routed, otherwise it is unset. In the latter case, if the `Reliable` field is set, then the corresponding `OMAC` and `ID` are stored internally. They are valid for limited amount of time, which is based on the circulation period. Along with those fields, two additional 1-bit status fields are stored, i.e. reception and routing, as well as a time-stamp field and its valid 1-bit field. Up to 3x16 of such 6-fields entries are stored and managed internally, i.e. based on the maximum number of circulating frames within a ring. While processing an incoming frame/packet, those 6-fields

entries are only used if the corresponding `Reliable` field is set. In other words, they are used if retransmission is a possibility.

When such a frame reaches the router port and the port's `RCL` bit is unset, it is then checked against the stored and still valid 6-fields entries. If there is a match, then it is considered a retransmission and routing/reception is done based on the corresponding status fields within its entry. Thereafter, the port's bit within the `RCL` field is updated accordingly. On the other hand, if either there is no match or the matched entry is invalid, i.e. time limit exceeded, then an entry is allocated based on the frame's `OMAC` and `ID` fields. And the respective reception and routing status fields as well as the `RCL` bit are set accordingly.

In case of a router being a member but not responsible for routing the multicast packet, then the routing decision is ignored in setting the port's bit within the `RCL` field, i.e. routing status always has a bit value of one. Similarly, if the router is not a member but responsible for routing the multicast packet, then reception decision is ignored.

As for identifying the sender of a routed multicast packet, it is left to the user application to include the sender's address as part of the payload, i.e. similar to routed unicast packets. However, the number of bits used can be adapted to the actual network size. In other words, if the network is composed of less than 2^7 rings and the biggest ring within the network has less than 2^3 nodes, then not all 12-bits are required to store the sender's address within the payload.

After routing, the router does not convert the multicast frame into a transport frame as is the case in unicast routing. Thus, the same multicast packet can be routed multiple times if the routers within a ring have overlapping multicast routing table entries. Therefore, an additional frame type is introduced and used specifically to account for such a scenario, it is called `Multicast Transport`. It is the 9th frame type, i.e. does not extend the `Type` field size. Furthermore, it does not affect or alter any data-link layer functionality. Indeed, it is processed in exactly the same way as `Multicast`. It is only used to signal to the router's network layer that the corresponding multicast packet has already been routed. With this method, the load

balancing approach explained in chapter 5 is not impacted.

Chapter 8

RTL Hardware Design and Implementation

This chapter provides a detailed description of the node and router designs mentioned in previous chapters as well as their implementation in hardware. The first section provides an overview of those designs and their components. Then, the design concepts, patterns and methods used within those designs are covered over two sections. Thereafter, the used memories are described. Finally, a detailed description of the hardware implementation of CarRing IV layers is provided.

8.1 Overview of the designs

8.1.1 Structure and main components

As mentioned in previous chapters, there are two designs that implement the protocol in hardware, node and router. In essence, those designs consist of four main parts that in turn contain the designs' main components. As shown in figure 8-1, the four parts are timer, protocol (sub-)layers, frame/packet arbiters and memory.

As described in chapter 6, the timer is used for time-sensitive operations and consists of one design component. Unlike the timer, only the functionality of the protocol (sub-)layers is covered in previous chapters. In what follows, a brief description of

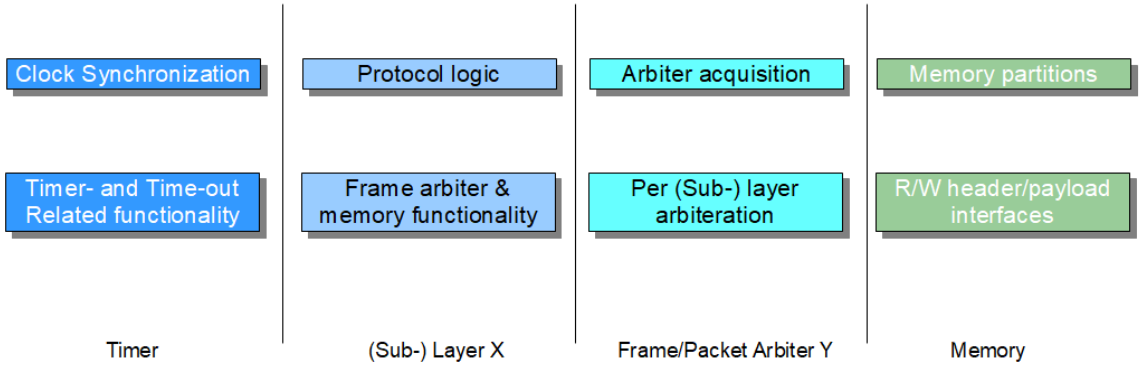


Figure 8-1: General structure of both node and router designs

the (sub-)layers' internal structure is provided.

Internally, each (sub-)layer is split into two sides, receiver and sender. Each side has its own design components. This split also applies to and affects the remaining two parts, frame/packet arbiters and memory. Regardless whether it is receiver or sender, each (sub-)layer-side consists of two main design components. The first component is named after the (sub-)layer-side itself and it contains the protocol's logic. As for the second one, it is an auxiliary design component that handles the frame/packet arbiter and memory interactions, which resulted from applying the protocol's logic. In other words, one component handles operations while the other commits their results.

As for frame/packet arbiters, they are responsible for frame acquisition and arbitration between operations within (sub-)layers design components. Each frame being processed is associated with a dedicated frame/packet arbiter and memory allocations. Before any frame processing can start, a frame/packet arbiter must be acquired and locked to the corresponding frame. In essence, a frame/packet arbiter maintains execution order with respect to protocol logic. It manages dependencies between operations within (sub-)layers, thus allowing operations with no inter-dependencies to run in parallel. Furthermore, it also manages access to the corresponding memory allocations.

Last but not least, memory is an essential part of both designs. It consists of one design component that divides the memory into sender and receiver side. In turn, each side is further divided into headers and payloads. Furthermore, it also includes

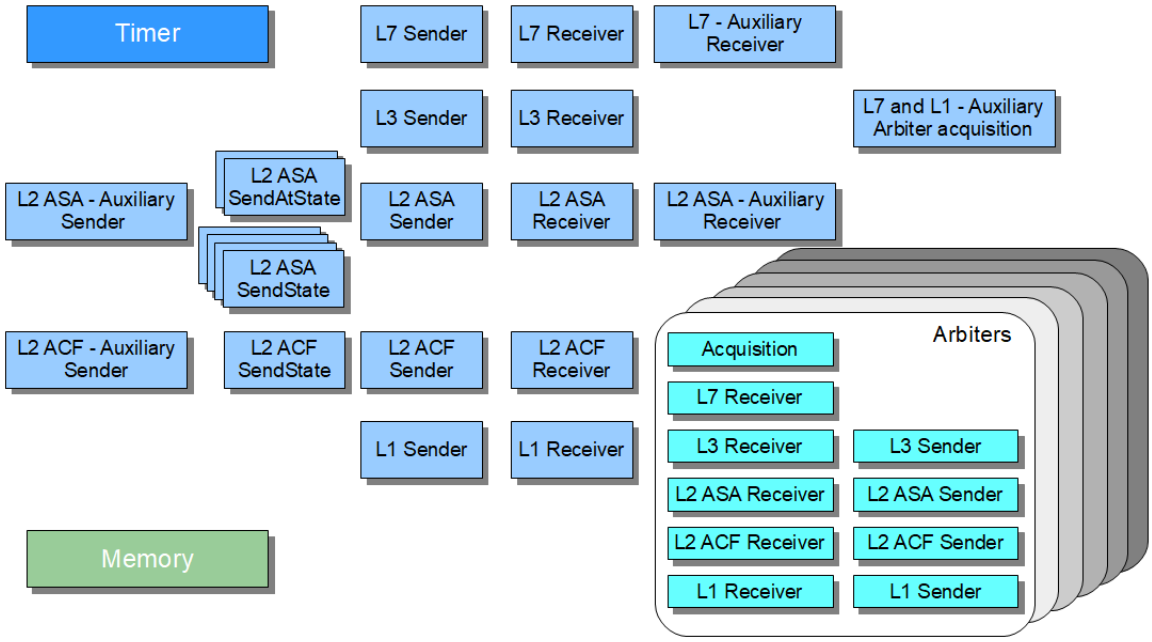


Figure 8-2: General structure of node design

read and write interfaces that are specific to each of the aforementioned allocations.

8.1.2 Node design

As shown in figure 8-2, not all (sub-)layers have an explicit secondary design component, i.e. an auxiliary, within the node design. On the other hand, some have additional design component(s) as well as an auxiliary. In what follows, a brief description of the aforementioned cases as well as the multi-layer auxiliary, i.e. L7 and L1 arbiter acquisition, is provided.

As described in the previous section, auxiliaries handle frame/packet arbiter and memory related operations. In certain (sub-)layer-sides, more clock cycles are consumed by the communication channels between the (sub-)layer-side main design component and the auxiliary, than saved due to the introduction of an explicit auxiliary design component. This is due to either the protocol logic within the (sub-)layer does not require multiple clock cycles, or there are too frequent interactions required with the corresponding frame/packet arbiter in order to apply the protocol's logic. In the latter case, the (sub-)layer's main component must frequently communicate

with the auxiliary to get feedback from the frame/packet arbiter before concluding any protocol logic operation.

Within the sender-side of the data-link layer, additional design components are introduced. In case of the ACF sub-layer, the `SendState` design component is used to handle and maintain the order of send requests received from higher layers. As for the ASA sub-layer, four `SendState` design components are introduced. They are used to manage and track the progress of acknowledged transmissions as well as reliable multicast transmission. However, they are only used if the corresponding `Retry` or `Reliable` bits are set, i.e. if retransmission is a possibility. Similarly, two `SendAtState` design components are introduced to manage and track periodical transmissions as well as at a future point in time, i.e. delayed transmissions.

As for the L7 and L1 arbiter acquisition auxiliary, it is a separate design component that is dedicated to acquiring arbiter frames on behalf of both, the sender-side of the application layer as well as the receiver-side of the physical layer. This is due to those two layers being the source of new incoming send requests and frames, respectively. The acquisition is performed automatically, i.e. without the need for an explicit request from the aforementioned layers. Otherwise, additional processing delay would be introduced. A new acquisition is triggered if the corresponding layer uses the current acquired frame/packet arbiter.

8.1.3 Router design

As described in chapter 5, a router includes two ports each with its own layers. In figure 8-3, the dotted lines indicates where direct duplication of the layers within the node design ends. Despite having two network layers, those layers are not direct duplicates of their node design counterparts. Within each network layer, the receiver-side directly interacts with the sender-side of the ACF sub-layer within the other port. Such interaction is part of the network layer routing functionality, i.e. issuing route requests. In short, a route request is essentially a send request issued by the other port's network layer instead of the application layer.

Furthermore, each memory design component includes additional interfaces for the

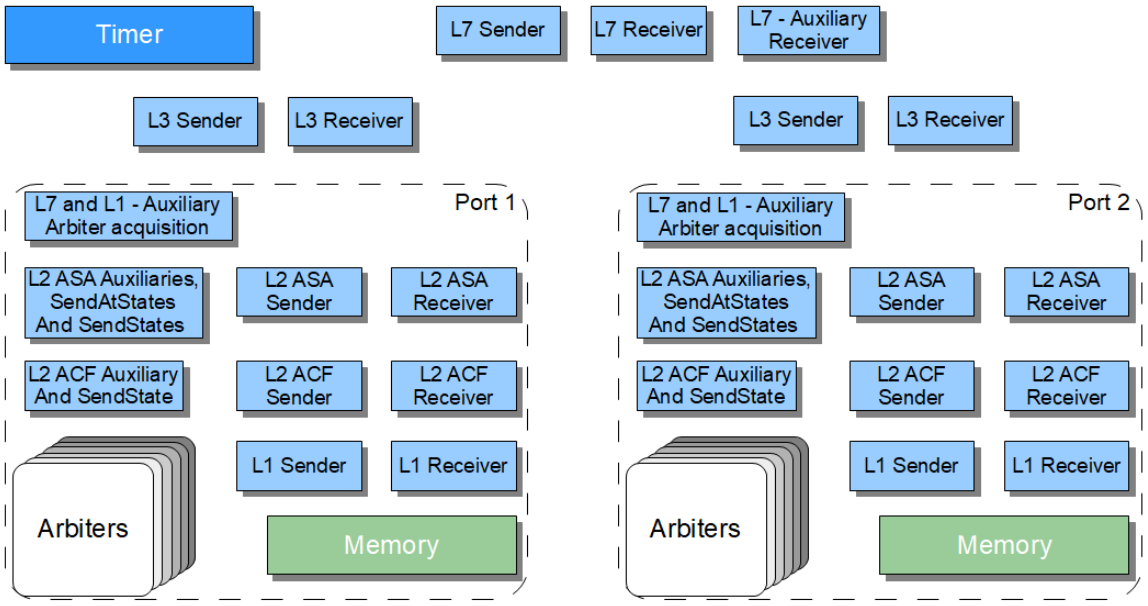


Figure 8-3: General structure of router design

router's other port. More specifically, it includes read interfaces for the receiver-side as well as the sender-side of the other port's network and physical layers, respectively. As for the application layer, it includes twice the number of user interfaces that of its node design counterpart.

8.2 Design concepts and patterns

8.2.1 Frame and packet processing

In order to benefit from the hardware's inherit parallelism, protocol layers are designed as PEs. Each PE only handles the protocol logic operations associated with one (sub-)layer. More specifically, there is one PE per receiver- or sender-side of a (sub-)layer. Those PEs do not store or track any frame or packet state information. Furthermore, they also do not include any memory operations. As described in section 8.1.1, such operations are encapsulated, along with frame/packet arbiter interactions, in an additional auxiliary design component.

As for the frame and packet state information, they are managed by frame/packet arbiters, which are described in the sub-section below. Moreover, there are other

state information that must be stored and managed long after the corresponding frame/packet has been processed and sent into the ring. This is required in case of acknowledged and reliable multicast transmission as well as the SendAt feature. Such state information is managed by dedicated SendState and SendAtState components.

While processing, a PE does not temporarily store the corresponding frame or packet. In other words, a frame/packet is not transferred through all layers until it reaches the application layer during reception or vice versa during transmission. Rather, only a multi-byte word along with a memory address-like value, called frame index, is exchanged between (sub-)layers. The frame index is covered in sub-section below. This multi-byte word contains only the part of the processing outcome that is relevant for the next (sub-)layer. Thus, no payload is transferred, i.e. only a control interface is required between (sub-)layers. The size and internal structure of the aforementioned word depends on the corresponding (sub-)layers.

As described previously, the physical layer handles the encapsulation and decapsulation for all higher layers. This effectively separates the communication channels used to transfer control/processing data and frame/packet data, i.e. user data. In other words, PEs have their own communication channels that are not used to transfer payloads. Thus, avoiding unnecessary processing delays.

As for the control interface, it is essentially a small-sized FIFO equipped with two point-to-point interfaces for the respective PEs, i.e. TLM-FIFO interfaces. The interface data-width is equal to the combined size of the aforementioned frame index and multi-byte word. As for the FIFO's capacity, it depends on the corresponding (sub-)layers, where the largest one holds at most four. Using a TLM-FIFO rather than direct point-to-point interface, decouples and disentangles the corresponding PEs. Thus, the respective (sub-)layers can process incoming frames/packets asynchronously, i.e. as fast as possible. This frame/packet processing approach allows for horizontal scaling, where more PEs can be added per (sub-)layer. The approach falls within the category of protocol multi-processing.

8.2.2 Arbiters

Within both, node and router designs, (sub-)layers are essentially PEs that do not store or track state information while processing a frame or packet. Handling such information is delegated to a dedicated design component, called arbiter. In addition to state information, those arbiters also handle processing inter-dependencies between (sub-)layers. Thus, they manage the execution order with respect to the protocol logic within PEs. And consequently, access to memory. In other words, they synchronize operations carried-out in parallel by different PEs. This effectively removes the sequential processing that results from a direct implementation of a layered architecture.

There are multiple arbiters within the node and router designs. The number of arbiters is directly related to the number of partitions within the designs' memory. In short, memory is partitioned based on frames and their respective packets. Each frame/packet memory allocation is associated with a dedicated arbiter. Thus, in order to use an allocation for a new frame/packet, its arbiter must first be engaged. Correlating allocations with their respective arbiters is achieved by using a memory address-like value, called frame index. This index value is used by (sub-)layers to select an arbiter's interface as well as calculate the corresponding memory addresses. In other words, it is used to distinguish between arbiter and memory allocation pairs.

As depicted in figure 8-2, an arbiter consists of multiple sub-components that not only interact with their respective (sub-)layers but also internally with one another. Since a new frame/packet originates either from the physical layer's receiver-side or the application layer's sender-side, arbiter acquisition must be managed between those two layers. Therefore, the arbiter also includes a sub-component that is dedicated to handling acquisition requests in addition to having a sub-component for each (sub-)layer. Such requests take place during the network's normal operation. However, that is not the case during ring initialization. More specifically, the creation and handling of a `Reset` frame constitutes a special case, which is covered in section 8.4.

Depending on the origin of the frame/packet, its processing is concluded, if its

either sent into the ring or the respective payload is received by the user application. Once processing is concluded, the corresponding arbiter is released, i.e. the memory allocation along with its arbiter can be used to process another frame/packet. However, those two must be preserved if retransmission is a possibility. More specifically, if `Retry` or `Reliable` fields are set in case of acknowledged transmission or reliable multicast. Therefore, the arbiter acquisition sub-component additionally includes a locking feature. This feature allows locking the memory allocation and its arbiter long after the corresponding frame/packet is sent into ring. And thereby preserving the payload in case of retransmission.

8.2.3 Transaction-level modeling

From the early phases and throughout the development life cycle, both node and router designs relied in no small part on the transaction-level modeling approach. As described in chapter 2, it elevates the communication details between design components to a higher level of abstraction, thereby separating communication from computation. Design components exchange data in the form of transactions through an abstract communication channel.

Within the context of CarRing IV, the majority of inter-process communication as well as interactions between design components use TLM interfaces. This partially reduces the complexity of the hardware implementation of both designs. While this preexisting modeling approach is not specific to one HDL, the aforementioned interfaces were mainly realized within SystemC, which is one of the main three HDLs used in CarRing IV. In what follows, the interfaces used in both designs are covered.

With the exception of memory interfaces, the majority of TLM interfaces internally use the standard ready/valid protocol. Figure 8-4 depicts the signals as well as the corresponding timing diagram of the handshake process. The valid and data signals are driven by the sender-side, while the ready signal is driven by receiver-side of the communication channel. Within the context of CtoS-TLM, the communication channel used in both the node and router designs, is called put/get channel. It internally uses the ready/valid protocol and is part of the Cadence Flex Channels

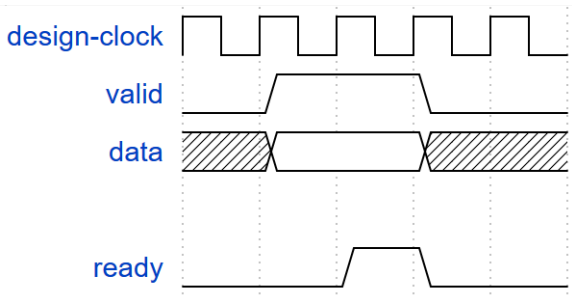


Figure 8-4: Timing diagram of valid/ready protocol

Library[15].

Despite sharing the same timing diagram at the signal level, there is a variety of interfaces used in both designs. They are differentiated based on their behavior[15]. In what follows, a transaction-level description of the interfaces and their behaviors is provided.

- **blocking put**. This interface always consumes at least one clock cycle before attempting to push new data into the communication channel. After one clock cycle, if the channel is not ready, then the interface continues to wait, i.e. consumes clock cycles until it is ready. Within the context of the designs, no operation can take place until the interface has pushed the corresponding data into the channel, i.e. operations are blocked once the interface is engaged.

- **non-blocking put**. Unlike the previous one, this interface does not consume any clock cycles nor does it ensure a successful data exchange across the communication. Rather, it delegates the responsibility to the design component, i.e. the process that is using the interface. The corresponding process is responsible for checking the channel before attempting to push data via the interface. Within the context of the designs, this interface is used when additional operations can be carried-out while waiting for the channel to be ready.

- **blocking get**. This interface is the counterpart of **blocking put**. It exhibits the same behavior, however on the receiver-side of the communication channel.

- **non-blocking get**. Similarly, this interface is the counterpart of **non-blocking**

put.

- **non-blocking get-peek**. In addition to the non-blocking behavior, this interface also allows a process to essentially read the channel data without completing the handshake process of the ready/valid protocol. Thus, a process is able to peek at the channel data before fetching it from the channel and thereby freeing the channel to accept new data. Within the context of the designs, this interface is used to check for and select an outcome from multiple different processes.

Last but not least, memory interfaces created within the designs are based on TLM constructs, i.e. *sc_port*, *sc_export* and *sc_interface*, which are described in chapter 2. At the signal level, they are essentially represented by standard memory write interfaces as well as synchronous and asynchronous memory read interfaces.

8.2.4 Token-based data-flow

So far, the structure, internal design components and communication channels of both node and router designs as well as the frame and packet processing approach are covered. This section covers the approach with which arbiters handle and process frame and packet state information. This includes the interactions of the arbiter sub-components with one another as well as with their respective (sub-)layers.

The approach can be best described as token-based data-flow. The execution of protocol logic operations is driven by the frame and packet data, i.e. headers' fields. Two types of tokens are used, state and flow tokens. While both types have fixed sizes, they do not have a fixed internal structure, i.e. the values carried by the tokens depend on the respective arbiter sub-component as well as the corresponding (sub-)layer. Both types are implemented using the put/get channel described in the previous sub-section. With respect to the TLM interfaces used, the majority are blocking put/get interfaces and to a lesser extent non-blocking put/get interfaces.

For a better grasp of the necessity and context of the two types of tokens, consider the following example. In case of multicast transmission, the incoming frame must be simultaneously received by the user and pushed back into the ring. Thus, the

corresponding flow of this incoming frame is accept and forward. However, its current state depends on the processing progress, i.e. still being received by user or being pushed back into ring.

The state tokens are exchanged internally between arbiter sub-components. They are mainly 1-bit in size with few consisting of 3-bits. Their main purpose is to exchange and update state information within each arbiter sub-component. The updates are invoked by the processing outcomes of the corresponding (sub-)layer.

As for the flow tokens, they are exchanged between arbiter sub-components and their respective (sub-)layers. They are 9-bits in size and consist of the processing outcome of the respective (sub-)layer as well as the flow information. Such information is represented by a flow type data structure. Due to the fixed number of possible headers' variations, 31 flow types are identified and implemented.

Flow types: `Default, Forward, Ring init, Acknowledge, Reuse, Forward Accept, Forward Not Accept, Release Transport Frame, Send Request, No Send Request, Send Request Accepted, Send Request Rejected, Process Send Request, Positive Acknowledge, Negative Acknowledge, Receive, Release Arbiter, Acknowledged, Circulation Period, Process Payload, Immediate, Create Transport Frame, Create Transport Frame And Send, Send, Retry, Receive Acknowledged Frame Type, SendAt, Time-out, Route, Clock sync, Forward Accept Multicast.`

In both node and router designs, the internal flows are directly affected by frame and packet header fields. While the payloads are dynamically set by the user, the data input that drives the internal flows within the designs is not dynamic, i.e. fixed and limited number of possible combinations of header fields' values. Therefore, it is not required to monitor and resolve deadlocks or livelocks during run-time. Rather, they are detected and resolved before synthesis, i.e. before deployment.

Access to shared memory is indirectly managed via both token types. In other words, managing the execution order of the protocol logic operations also takes into account access to shared memory. Thus, there are no explicit tokens or design components that directly manages access to shared memory. This moves the problem of

shared memory access management from a central design component, which is the classical or typical approach, to distributed components that manage the execution of protocol logic operations.

8.3 Memory management

There are two types of memory used within node and router designs, Flash memory and BlockRAM. The former is a separate and dedicated component within the prototype, while the latter is the FPGA's internal memory. Within the Kintex7-prototype, this dedicated component is the S25FL256SAGBHI20 chip. As for the Virtex5-prototype, the PC28F256P30T85 chip is used.

A node or router configuration data are manually set within the prototype's Flash memory before run-time. After power-up, the configuration data is immediately fetched from flash memory and used internally within the designs, i.e. before ring initialization. The use of flash memory to set configuration data is optional. In other words, the configuration interface in both designs can be used and adapted to custom user designs. As such, the use of flash memory is regarded as more of a convenience rather than a necessity.

On the other hand, BlockRAM is used internally by both designs for processing frames and packets. When compared to general purpose processors, it is the equivalent of an L1-cache. It operates at the same clock frequency as the designs. Thus, no unnecessary delays are encountered, when interacting with the memory component within both designs. Using a standard memory solution, e.g. DDR-X, would require a differential clock signal to drive the memory component. This standard approach introduces another clock domain into the designs. Consequently, each memory interaction includes additional overhead due to clock domain crossing. Thus, it is considered sub-optimal for both designs.

Within both node and router designs, memory is partitioned based on frames and packets, i.e. divided into headers and payloads. As show in figure 8-5, the memory is further divided to match the (sub-)layers' general structure, i.e. sender- and receiver-

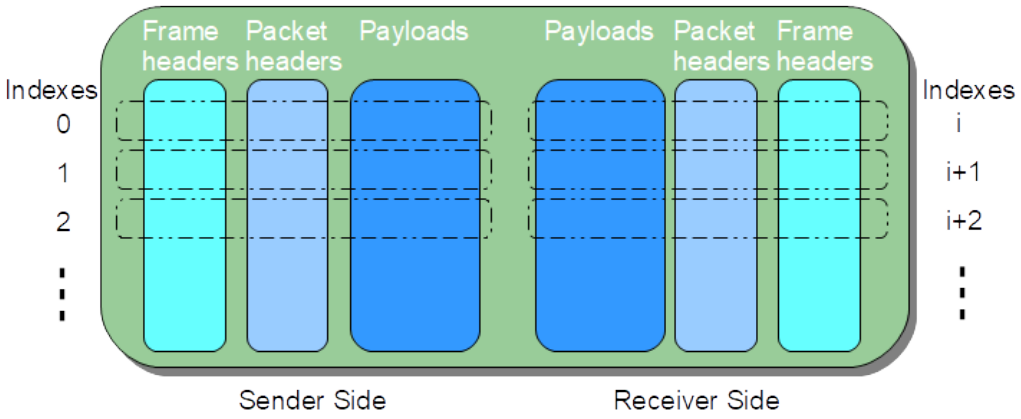


Figure 8-5: Internal structure of main memory

side. As for memory interfaces, the corresponding TLM interfaces used by (sub-)layers mask the internal memory structure. In other words, (sub-)layers use one read and one write TLM interface to interact with the memory component. This is achieved in-part by using the aforementioned frame index.

As described previously, the frame index is used to correlate arbiters and their corresponding memory allocations. This correlation is not dynamic, i.e. each arbiter has its own fixed memory allocation in-which the corresponding frame and packet resides. Therefore, the total number of frame indexes reflects the total number of frames and packets that can be stored and processed within the node and router designs. Thus, increasing a design's capacity to process more frames and packets implies additional arbiter/memory-allocations pairs.

In order to both mask the internal memory structure and accommodate the sender- and receiver-sides of (sub-)layers, the frame indexes are split into two sets or ranges. The sender-side attempts to acquire arbiters using frame indexes belonging to its range and the receiver-side is similarly restricted to its range. More specifically, arbiter acquisition attempts are done by the application layer on the sender-side, and the physical layer on the receiver-side. The splitting of frame indexes is specific to arbiter acquisition and memory resources. In other words, an arbiter includes sub-components for both sides of all (sub-)layers. Thus, PEs from both sides can be involved in processing using the same arbiter and memory resources.

While the frame index is used by (sub-)layers' auxiliary component to select an arbiter's interface, it is also used to calculate memory addresses. Since the arbiter interface selection method is straight forward, in what follows, the use of frame indexes within the context of memory addresses is described.

The memory read and write TLM interfaces are associated with a data structure, called memory request. It consists of the three fields, `Frame index`, `Address offset` and `Data`. The frame index is used to check for the origin of the memory request, thereby which side of the memory is engaged next. As for the address offset, it is used to navigate through the headers or payloads memory partitions. As shown below, the memory address is calculated using both the frame index and address offset fields within the request.

$$HeaderOrPayloadAddress = FrameIndex \times HeaderOrPayloadSize + AddressOffset \tag{8.1}$$

Last but not least, the `Data` field is used within the context of a memory write. With respect to payloads, the size of both this `Data` field and the returned data in case of a memory read is equivalent to the data-width of Aurora's LL interface. As for headers, the size is set to 64-bits.

8.4 Hardware implementation

In this section, the descriptions provided are specific to implementation. They are considered as complementary to the functionality sub-sections in chapter 5. This section covers the implementation, internal workings and behavioral specifics of the node and router designs. To avoid bloating this section with a line-by-line listings of the used HDLs, i.e. design codes, in what follows, the descriptions are provided using flow diagrams and layouts figures and depictions.

In order to avoid cluttering the figures used below with legends and additional descriptions, the used symbols and elements are briefly described. As shown in figure

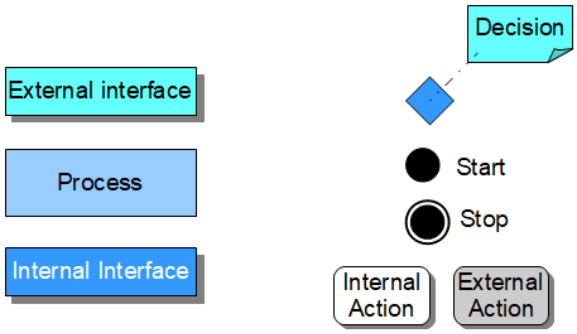


Figure 8-6: Illustration of layout symbols and flow diagram elements

8-6, there are two interface representations used. The external interface corresponds to input and output ports that are external to the node and router designs. For example, configuration ports that originate from pins corresponding to a physical button or switch within the prototype. Another example is Aurora's LL interface. Thus, they exclusively represent input and output ports as opposed to internal interfaces. Such interfaces originate from and connect design components within the node and router designs.

As for flow diagrams, the start, stop and decision elements are self-explanatory. On the other hand, the action elements correspond to logic operations performed within arbiter sub-components. Such actions are considered internal if they originate from interactions that are internal to the arbiter, i.e. using TLM interfaces between arbiter sub-components. And they are external if the corresponding interactions are with other design components external to the arbiter.

Last but not least, TLM interfaces are effectively implemented using signals i.e. regular I/O ports. However, within the design their transaction level implementation is referenced. Several shortcuts are used in-text and figures to reference TLM interfaces. Those are:

- nb_put, b_put: non-blocking and blocking put interface
- nb_get, b_get, nb_get_peek: non-blocking, blocking and non-blocking peek get interface

8.4.1 Node

Reset logic and configuration data

As described in chapter 2, reset logic is an important part of any design. Within the node design, synchronous active high reset is used. The reset logic is based on three signals. The first is the output signal derived from debounce logic associated with a typical reset button. As for the second signal, it is the output signal of a *VHDL process* that monitors the channel-up output signal of Aurora. Such a signal is set once Aurora has finished initializing. And the monitoring is performed for a limited number of clocks, after-which the Aurora channel is considered stable and ready for use. Last but not least, the third signal is a configuration done signal, which indicates that the node configuration data has been properly set.

A side effect of this reset logic is that the node design starts in a reset state for multiple clock cycles before transitioning into operational state. Thus, giving the node design enough time to initialize internally used resources, e.g. interfaces, signals. This is crucial to ensuring a functional node design.

As for configuration data, a separate and independent *SC_MODULE*, called NodeConfig, is used to provide and set the node configuration data. It fetches the data from flash memory. And using the node's configuration interface, it sets appropriate input ports. The aforementioned configuration done signal is only set after all data has been retrieved and set. The configuration data includes:

- Primary enable input. As name implies, it resembles whether the node is a primary node, i.e. whether it is responsible for ring initialization.

- Reference clock enable input. It is only relevant if the node is a primary. As part of the clock synchronization, it indicates whether the corresponding primary node is also the reference node for the network.

- Node's ring and MAC addresses. Those are used only if the node is a primary node.

- Ring initialization time-out. Similarly, this value is only used if the node is a primary.
- Mode of operation. This input indicates whether the isochronous or anisochronous mode is used.
- Multicast membership table.

Physical layer

Regardless whether it is the node or router design, the design-clock is a *single-ended clock* signal that is generated by Aurora. In other words, both designs operate at the same frequency and in the same clock domain as Aurora's user interface, i.e. LL interface. As a result, the frequency of such a clock must be adequate for the design. As described in chapter 2, the clock period of the used clock signal must not be smaller than the design's lower limit. Otherwise, the design's timing constraints can not be met, i.e. negative slack. Therefore, Aurora's configuration must accommodate both designs. On the other hand, the physical layer and the frame size is in-part based on Aurora's frame transmission behavior.

Since Aurora's internal workings and specifics are outside the scope of this thesis, in what follows, only the parts that directly affects the designs are briefly described. In short, Aurora's configuration is based on number of lanes, bytes per lane, line rate and the reference *differential clock* (aurora-clock). Based on the configured values, the clock period of the aforementioned design-clock is calculated. To a certain extent, the configuration is in-part covered in sub-section 5.1.4, i.e. number of lanes, bytes per lane and additionally the design-clock.

As for line rate and aurora-clock, those are based on the used network cables and prototype's clock generator chip, respectively. In case of Virtex5-prototype, this translates to one lane per port with 3.125 Gbps and a 250 MHz aurora-clock, whereas it is four lanes per port with 6.25 Gbps and a 125 MHz aurora-clock for Kintex7-prototype.

In essence, Aurora is another design component with its own timing constraints

and sub-components. As such, there is latency associated with frame transmission within Aurora. It stems from Aurora's protocol engine as well as the underlying GTX transceiver. This results in a pipeline effect. Thus, the designs must continuously transmit, i.e. fill the link to achieve the highest throughput possible.

At the early stages of the project i.e. Virtex5-prototype, hardware level loop-back tests were conducted to determine the number of aurora-words required to fill the link, thereby determining an adequate frame size. Those loop-back tests were external to the transceiver, i.e. the data was transmitted onto the network cable and captured back again on the receiver interface using Xilinx Integrated Logic Analyzer (ILA) IP core. As for aurora-word, it is the number of bits per one clock cycle pushed through the LL interface, i.e. LL's data-width, which in turn is based on the number of bytes per lane and number of lanes. The end result is that the frame size is set to 16 aurora-words, where the link can hold up to 2 frames.

Note that while it is also possible to fill the link using fewer but larger frames, the increased number of frames results in more sending opportunities, thereby a more responsive system.

In case of Virtex5-prototype, an aurora-word is 32-bits in size which translates to a 512 bits frame. Due to the increased number of lanes within Kintex7-prototype, an aurora-word is 128-bits in size which results in 2048 bits frame. Within CarRing IV, the main focus is on the timing behavior of the protocol rather than the number of bits in a frame. In other words, the number of clocks consumed to push a frame via Aurora's LL interface, rather than the frame's size. As such, the 16 clocks per is preserved as the project transitioned from the old prototype to the current one.

As depicted in figure 8-7, not all layers require explicit auxiliary design components. Due to the layer's functionality, the PE and auxiliary component are merged. This results in one design component per sender- and receiver-side. In case of the sender-side, one *SC_CTHREAD* is sufficient to push frames to Aurora via the LL interface. It gets the frame index from the upper layer, i.e. the ACF sub-layer. And it uses the frame index to fetch data from memory as well as interact with the corresponding arbiter using the appropriate *b_put/b_get* interface pair. Note that NoF

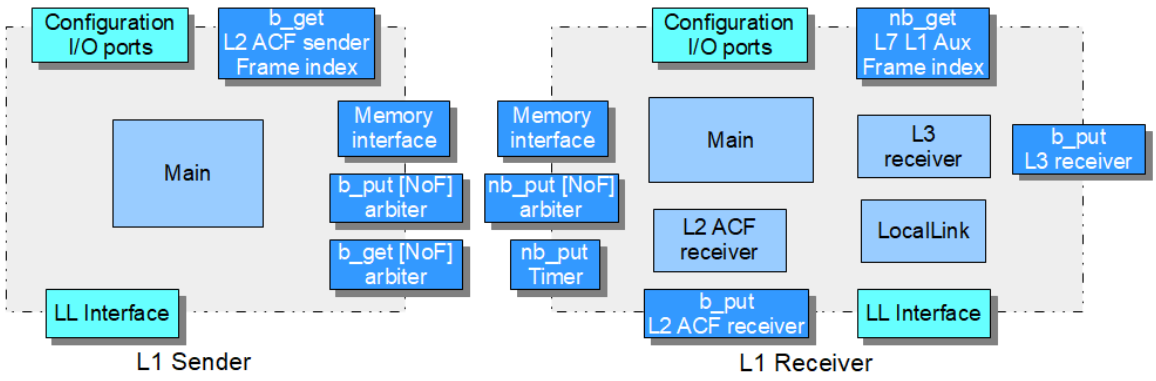


Figure 8-7: Physical layer sender- and receivers-side designs

stands for number of frames which represents the design's capacity in terms of frame and packet processing, i.e. number of arbiter/memory allocation pairs.

On the other hand, the receiver-side requires multiple *SC_CTHREAD*. Although it is possible to pause data reception via the LL interface, this is explicitly avoided in order to achieve the highest throughput possible. Therefore, as an additional measure three more *SC_CTHREAD* are introduced to avoid the possibility of a temporary pause. As the frame is being received, the headers are pushed to the corresponding (sub-)layers via dedicated *b_put* interface and *SC_CTHREAD*. As for the frame index, it is acquired from the L7 L1 arbiter acquisition auxiliary prior to data reception. Similar to the sender-side, the frame index is also used to select and interact with the corresponding arbiter and memory allocation.

As described previously, the frame and packet state information are handled by arbiters. As shown in figure 8-2, each arbiter sub-component corresponds to a (sub-)layer, e.g. the sender-side of the physical layer has a dedicated sub-component, called Arbiter-L1-S. The general structure of a sub-component is the same across all (sub-)layers. In short, there is one *SC_CTHREAD* that receives flow tokens from the corresponding (sub-)layer and exchange state tokens with other arbiter sub-components, i.e. other (sub-)layer specific arbiters. Such interactions are performed using *b_put*, *nb_get* and *b_get* interfaces. Thus, rather than including design depictions that are mostly similar, flow diagrams are used to explain the internal workings of arbiters. This applies for all (sub-)layers, including the router design sub-section.

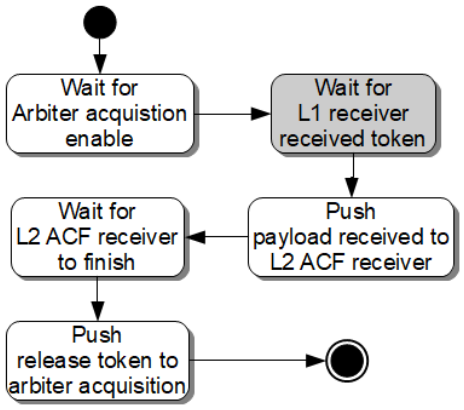


Figure 8-8: Flow diagram of physical layer receiver-side arbiter

Figure 8-8 depicts the flow diagram of Arbiter-L1-R. On the receiver-side of the design, frames are first encountered in the physical layer. Therefore, it is responsible for acquiring arbiter and memory allocation pairs on behalf of all layers within the receiver-side of the node design. As depicted in the diagram, the first action is to wait for arbiter acquisition sub-component to confirm acquisition, i.e. enable and start. Thereafter, the corresponding frame index is pushed to and used by (sub-)layers of both sides of the design. During decapsulation, the frame index is pushed to higher layers. Thus, Arbiter-L1-R is required to only inform higher layers that the frame is fully received, i.e. payload is received.

In order to reduce arbiter sub-component interactions, each sub-component interacts with its immediate neighbors, i.e. with sub-components belonging to the (sub-)layer directly above or below. Therefore, waiting for ACF receiver sub-layer, depicted in figure 8-8, is affected by the processing outcome of higher layers. Thus, the wait can be based on waiting for user to fully receive the payload, and/or waiting for the sender-side to push the frame back into the ring.

Unlike the receiver-side, arbiter acquisition is performed by the application layer within the sender-side. As depicted in figure 8-9, the starting point for Arbiter-L1-S is waiting for the higher layer, i.e. ACF sub-layer to finish processing the header. Note that one of the paths in the flow diagram leads an immediate arbiter release shortly after starting. This is possible within the context of processing a user send request. In such a scenario, there are two arbiter/memory allocation pairs, one for the incoming

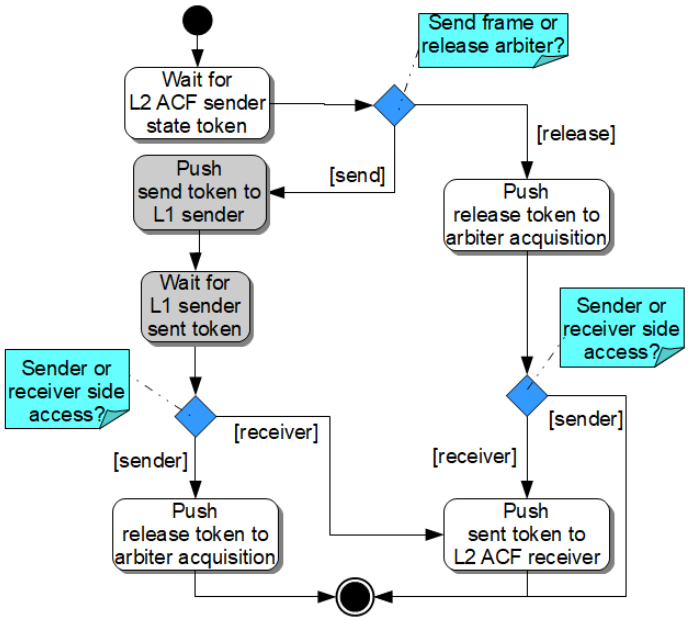


Figure 8-9: Flow diagram of physical layer sender-side arbiter

frame and another for the send request. Rather than copying data internally between memory allocations, only the header of the incoming frame is reused, while the rest is released. Thus, only one arbiter/memory allocation pair remains active, i.e. one associated with send request. Consequently, only one frame is sent.

Last but not least, the protocol operation modes, i.e. isochronous and anisochronous, are not explicitly accounted for within the arbiter sub-components. There are no input/output ports or interfaces used to exchange or check for which operation mode is active. Rather, operation modes implicitly affect arbiter's external interactions with (sub-)layers as well as internal interactions between sub-components.

ACF sub-layer

Unlike the previous sub-layer, the ACF requires a separate auxiliary design component in addition to its PE. As depicted in figure 8-10, there are two sources for frame indices, the receiver-side of either the ACF or ASA sub-layer. Depending on the frame type, the corresponding sub-layer is responsible for providing the frame index, i.e. ACF in case of unacknowledged and ASA for acknowledged. As for the *SC_CTHREAD*s, they are based on the sub-layer's functionality.

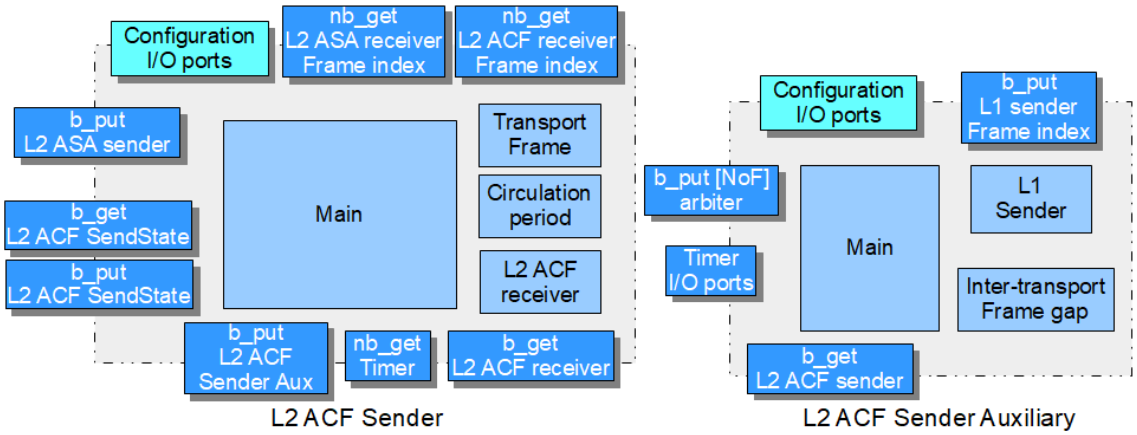


Figure 8-10: PE and Auxiliary designs of the ACF sub-layer sender-side

During ring initialization, the initial creation and transmission of `Reset` as well as a node's own `Transport` frames is considered as a special case. In such a case, an arbiter/memory allocation pair is not required, i.e. no arbiter acquisition is performed. Consequently, the usual interactions with an arbiter as well as a memory allocation is bypassed during transmission. The frame header is created internally by the sender-side of the ACF sub-layer. And the payload is essentially padded during transmission, i.e. one aurora-word repeated until the full frame size is reached. As for the packet header, it is also padded with default values.

Unlike typical approaches, user send requests do not result in the creation of a frame, which is then pushed into the network. Rather, they wait for a sending opportunity, i.e. for the node's own frame to circulate back or reuse a frame of another node. This maintains the fixed number of frames that circulate the ring, i.e. network. As such, another separate design component is introduced to maintain the order of user send requests internally within the node. It is called L2-ACF-SendState and is depicted in figure 8-11. This internal ordering is based on time of request arrival. It also covers retransmissions in case of acknowledged transmission which includes multicast. Note that NoS stands for number of send requests which represents the number of consecutive send requests received from the user.

As shown in figure 8-12, there is two possible starting points within Arbiter-L2-ACF-S. Either a state token from the receiver-side of the ACF sub-layer or the sender-

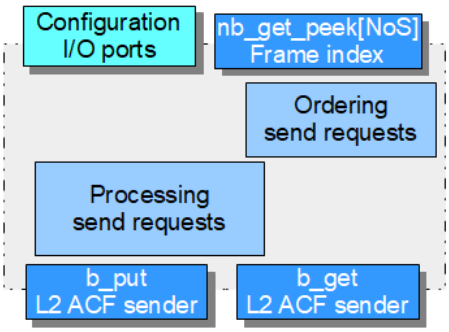


Figure 8-11: SendState design of the ACF sub-layer sender-side

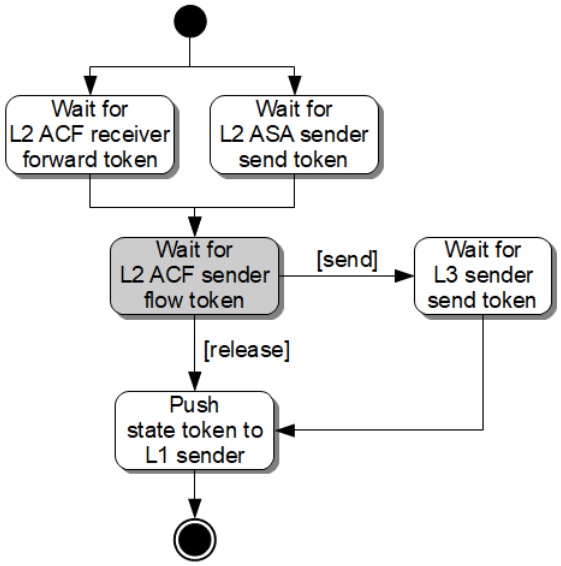


Figure 8-12: Flow diagram of ACF sub-layer sender-side arbiter

side of the ASA sub-layer. The former corresponds to an incoming frame that must be forwarded. And the latter corresponds to a user send request being transmitted, an acknowledge or a retransmission. As for the possible release flow token received from ACF, it corresponds to the user send request and incoming frame scenario described earlier, where only one the two arbiter/memory allocations pairs remains active and the other is released.

Figure 8-13 depicts the merged design of the ACF sub-layer's receiver-side. Unlike the physical layer, the corresponding PE and auxiliary are merged due to frequent interactions required to achieve the sub-layer's functionality. Thus, in order to reduce the overhead resulting from the frequent interactions, a merged design is used. As

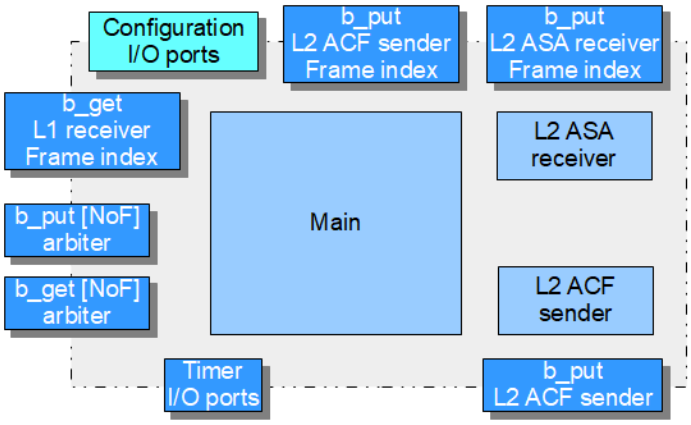


Figure 8-13: ACF sub-layer receivers-side design

for the frame index, it is received from the physical layer receiver-side. Based on the processing outcome, it is either pushed to ASA receiver-side or ACF sender-side. The former corresponds to an acknowledged frame type, where the corresponding payload is being received. As for the latter, either an unacknowledged frame is being simultaneously received and forwarded or simply forwarded.

Among the responsibilities of the receiver-side of the ACF sub-layer is handling the acceptance criteria. Therefore, the starting point for Arbiter-L2-ACF-R is waiting for the state token of the network layer's receiver-side, as depicted in figure 8-14. It contains the result of the ring address comparison, which is then used by ACF.

ASA sub-layer

In order to reduce processing complexity, two interfaces are used to receive frame indices from the network layer sender-side, as depicted in figure 8-15. The split is based on whether the frame index corresponds to an acknowledged or unacknowledged transmission. Since there is no retransmission in unacknowledged, the corresponding arbiter/memory allocation pair is released immediately after the frame is pushed into the ring. On the other hand, the headers and payload must be preserved in case of unacknowledged transmission. However, this is only the case if retransmission is explicitly enabled by the user, i.e. `Retry` field is set, or `Reliable` in case of multicast. Otherwise, the acknowledged transmission is treated as an unacknowledged one.

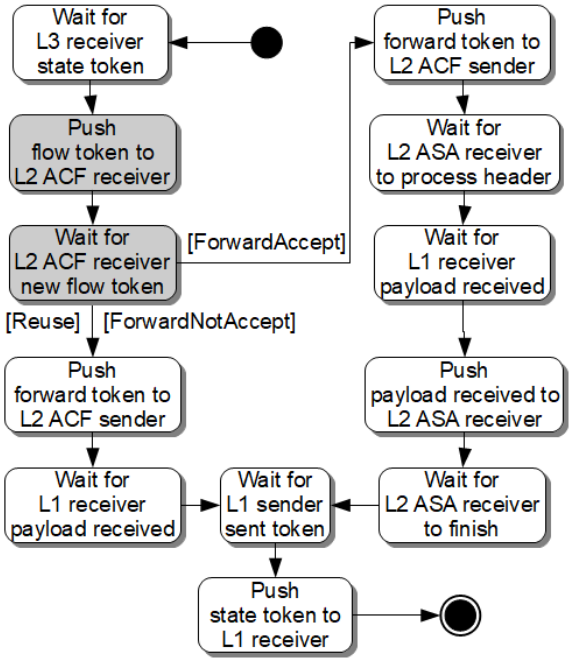


Figure 8-14: Flow diagram of ACF sub-layer receiver-side arbiter

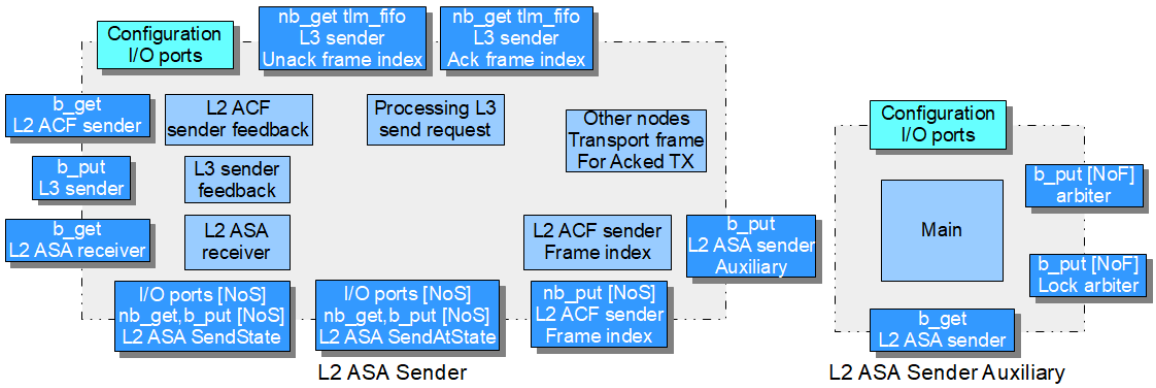


Figure 8-15: PE and Auxiliary designs of the ASA sub-layer sender-side

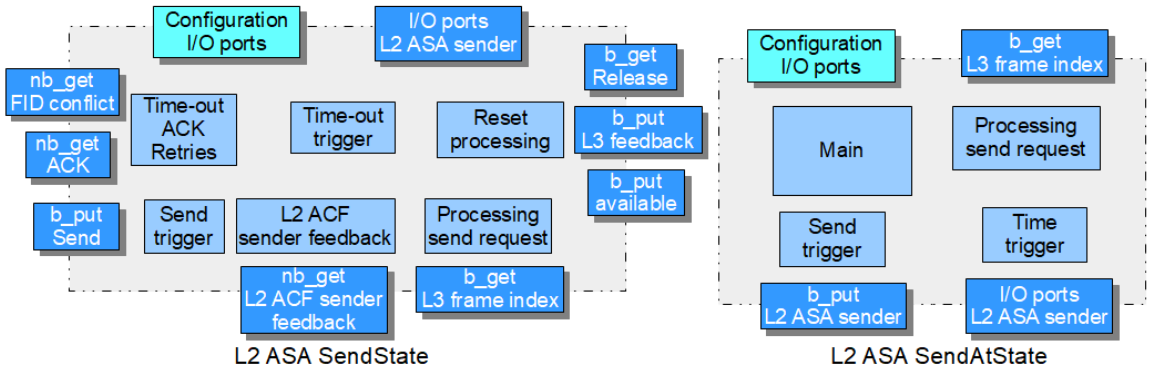


Figure 8-16: SendState and SendAtState designs of the ASA sub-layer sender-side

Additionally, there are two design components that encapsulate part of the sub-layer's functionality. The L2-ASA-SendState and L2-ASA-SendAtState are depicted in figure 8-16. As the name implies, L2-ASA-SendAtState handles the protocol logic operations pertaining to the SendAt feature. This one design component keeps track of and manages both periodical and delayed transmission.

On the other hand, the L2-ASA-SendState handles retries, time-out and incoming acknowledges for one user send request. Therefore, multiple duplicates of the component are used, where the total number is based on NoS, i.e. the number of send requests. As stated earlier, an acknowledged transmission is treated as an unacknowledged one if retransmission is not explicitly enabled. While this applies to the arbiter/memory allocation pair, it is not the case for the usage of an L2-ASA-SendState. Such a transmission still occupies an L2-ASA-SendState in order to keep track of time-out. However, a time-out trigger does not result in a transmission, rather it results in a notification pushed to the application layer.

Within ASA sender, the majority of the protocol logic operations correspond to retries, time-outs and acknowledges. And those have been delegated to specialized secondary design components, i.e. L2-ASA-SendState and L2-ASA-SendAtState. Therefore, the state information contained within Arbiter-L2-ASA-S is minimal, as shown in figure 8-17. It is mainly concerned with enabling ACF sender based on the flow token produced by ASA's PE.

Unlike the sender-side, the ASA receiver-side does not require additional design

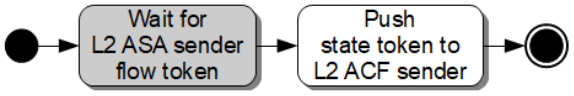


Figure 8-17: Flow diagram of ASA sub-layer sender-side arbiter

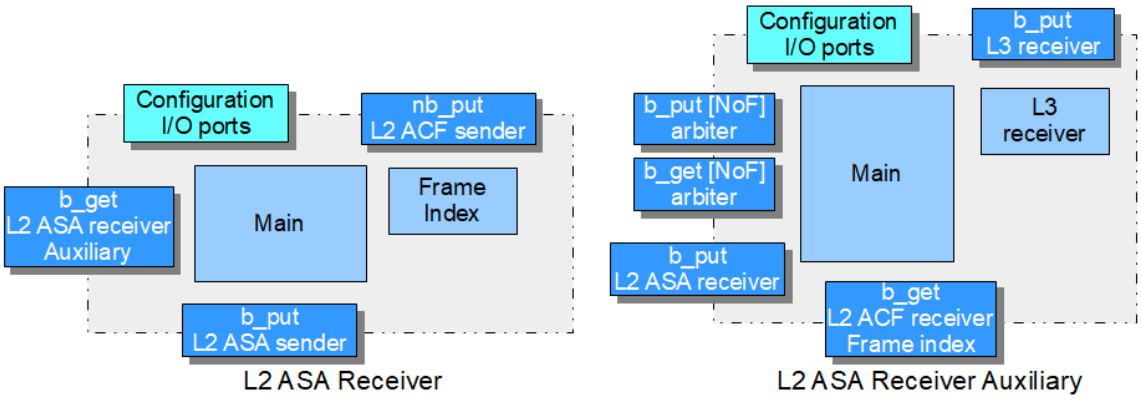


Figure 8-18: PE and Auxiliary designs of the ASA sub-layer receiver-side

components beyond PE and Auxiliary, as shown in figure 8-18. As a result, the state information is entirely contained within its respective arbiter.

The flow diagram of Arbiter-L2-ASA-R is depicted in figure 8-19. It also covers the case if application layer, i.e. the user is unable to read the payload in time. Depending on the frame/packet being received, this can result in a negative acknowledge in case of acknowledged transmission, or the `RCL` field not being updated in case of multicast.

Network layer

Similar to the physical layer, the layer's functionality is also relatively limited, when compared with other layers or with the network layer within the router design. As described in chapter 5, the functionality of a node's network layer is simplified, since it does not store, pull or push any routing information. In short, it is responsible for ring address, conversion of header fields from packet to frame as well as elevating some the data-link layer functionality and making it accessible to the application layer. It also handles the distinction between local and non-local destinations with respect to the i-to-j feature of the data-link layer.

Consequently, a merged design is used for both the sender- and receiver-sides of

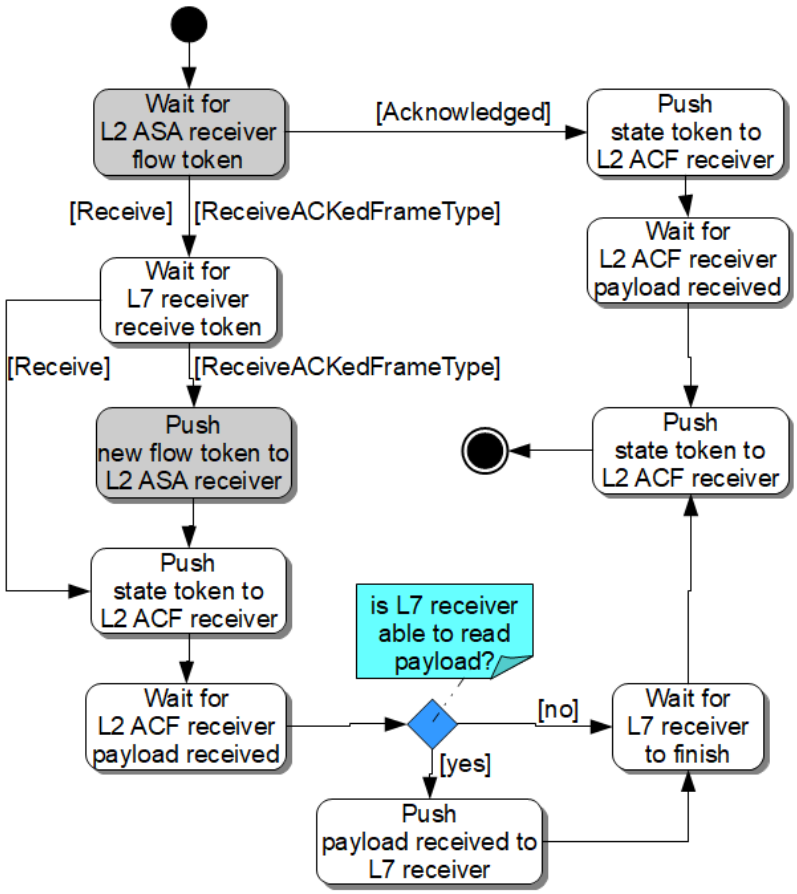


Figure 8-19: Flow diagram of ASA sub-layer receiver-side arbiter

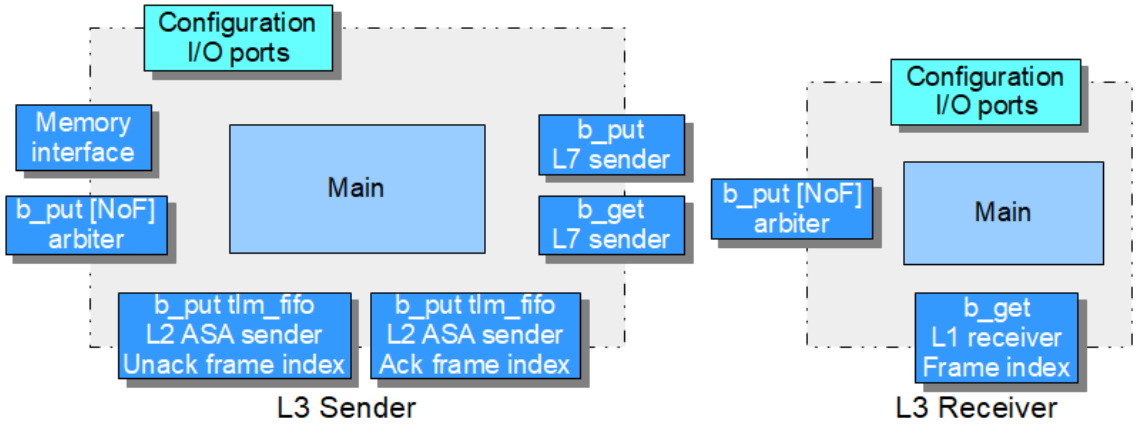


Figure 8-20: Network layer sender- and receivers-side designs

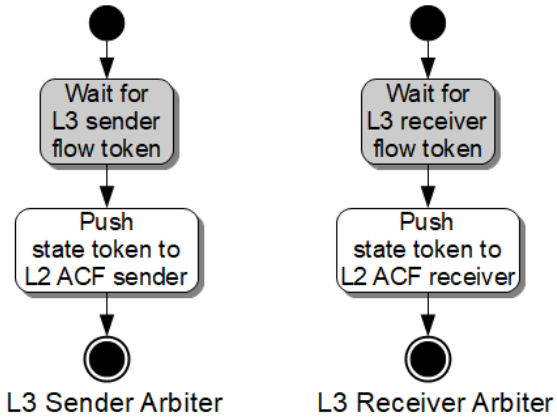


Figure 8-21: Flow diagrams of network layer sender- and receiver-side arbiter

the network layer. The designs are depicted in figure 8-20. The sender-side uses two interfaces to push frame indices to the ASA sub-layer. Those interfaces are the counterparts of the two ASA sender-side interfaces described above.

The reduced complexity and simplified network layer functionality is also reflected in the layer's Arbiters. The respective flow diagrams are shown in figure 8-21. On the sender-side, it revolves around determining the transmission type associated with a user send request. And on the receiver-side, it corresponds to the acceptance criteria within the network layer.

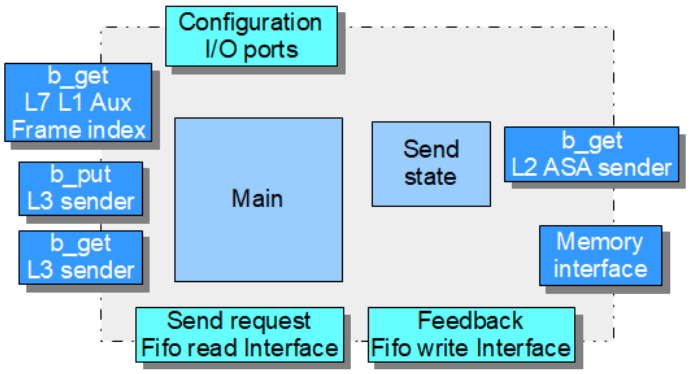


Figure 8-22: Application layer sender-side design

Application layer

Similar to both the network and physical layers, a merged design is also used for the sender-side of the application layer. However, unlike the other two layers, no arbiter interfaces are included in the merged design. As shown in figure 8-22, only the memory interface remains part of the design. Furthermore, no layer-specific arbiter is required for the sender-side of the application layer.

The sender-side is mainly responsible for handling interactions with user applications. In other words, no header manipulation is performed, thereby no Arbiter-L1-S is required. Nevertheless, the application layer's sender-side still receives a frame index from L7 L1 arbiter acquisition auxiliary. This is due to this layer being the source of new frames/packets within the sender-side of the node design. Therefore, it is responsible for acquiring arbiter/memory allocation pairs on behalf of all layers within the node design's sender-side.

As described previously, user send requests are not immediately transmitted. Rather, they wait for a sending opportunity, i.e. for an incoming frame that could in turn be used to transmit the corresponding payload. The eligibility of the incoming frame for transmission depends on multiple factors among which is the operation mode used, i.e. anisochronous or isochronous. However, the processing of the incoming frames is not interrupted, if a new send request is issued simultaneously.

More specifically, if the request is still being processed and has not yet reached the ACF sub-layer. Regardless of the transmission eligibility, the processing of the

incoming frame is not interrupted or paused to wait for the request. This design decision was made to maintain time predictability with respect to the arrival of frames, i.e. maintain deterministic behavior.

Standard FIFO read and write interfaces are used for interactions with user applications. In other words, a FIFO is used to transfer data between the application layer and the user application. Such an approach is used, because it offers a good balance between compatibility and flexibility. In other words, FIFO interface implementations are ubiquitous and an independent-clock FIFO can be used to cross clock-domains.

A data-width of 128 bits is used for the send request FIFO read interface. It is based on the data-width of an aurora-word, i.e. it is equal to data-width of Aurora's LL interface. As for the structure of the input data, 68 MSBs of the first input are reserved for the send request fields while the rest is used for the payload, i.e. 2008-bits. The request fields consist of: `Packet type` (3-bits), `destination MAC address` (4-bits), `destination ring address` (8-bits), `destination multicast address` (8-bits), `Retry` (1-bit), `Reliable` (1-bit), `SendAt index` (2-bits), `SendAt time interval` (40-bits) and `SendAt state` (1-bit).

With respect to the feedback interface, it has a data-width of 26 bits. And the feedback fields consist of: `Address status` (1-bit), `Payload status` (3-bits), `Frame index`, `RCL` (16-bits), `Frame ID` (2-bits) and `Frame Owner MAC address` (4-bits). The frame index field is required due to the increased number of sending opportunities within the anisochronous mode. Due to acknowledgments, unacknowledged send requests have more sending opportunities than acknowledged ones within this operation mode. As a result, an unacknowledged send request might be transmitted before an acknowledged one despite its time of arrival from the application layer. In such a scenario, the time of issuing a send request is no longer sufficient to match the feedback received with the request issued within the user application.

With respect to the status fields, the responses along with their respective integer values are:

- `Address status` (1-bit): `Invalid` (0) and `Valid` (1).

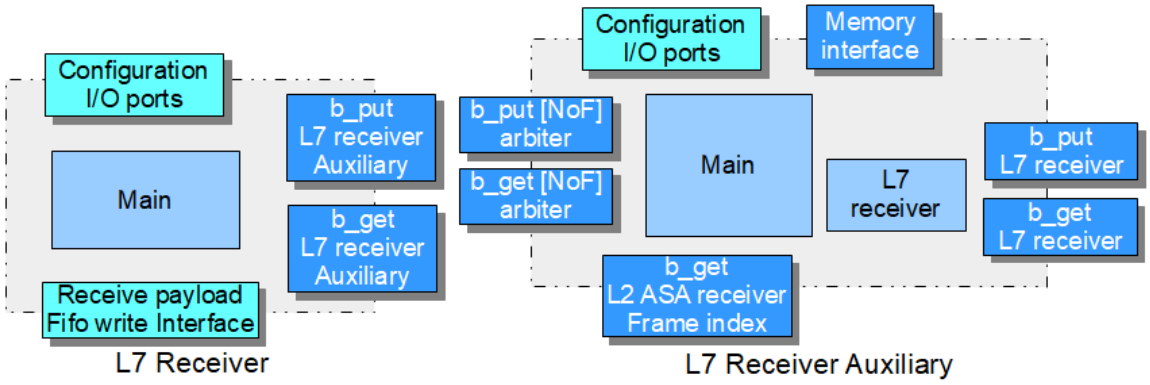


Figure 8-23: PE and Auxiliary designs of the application layer receiver-side

- `Payload status` (3-bits): `Received` (0), `Not received` (1), `Positive acknowledge` (2), `Negative acknowledge` (3), `Routed` (4) and `Not routed` (5).

As described in sub-section 5.4.3, the feedback includes a limited destination address check. However, there are no internal checks on the time interval supplied within the context of send-at request. More specifically, if a small interval is provided by the user application, then it is no longer a periodical or delayed transmission, rather it is sending as soon as possible, i.e. immediate with the next sending opportunity.

On the other hand, the receiver-side of the application layer consists of both a PE and an auxiliary design component, as depicted in figure 8-23. Unlike all other (sub-)layers, the frame index is received by the auxiliary design component instead of PE. This is due to no header manipulation being performed in the receiver-side of the layer. Thus, the frame index is used only to retrieve data from memory and update state information within the arbiter based on user application interactions.

Similar to the sender-side, the receive payload interface is also a FIFO write interface, where the data-width is 128 bits, i.e. the same as LL interface data-width. As for the structure of the output data, 42 MSBs of the first output are reserved for the reception fields, while the rest is used for the payload, i.e. 2008-bits. The fields consist of: `source MAC address` (4-bits), `source ring address` (8-bits), `source multicast address` (8-bits), `RCL` (16-bits), `Frame ID` (2-bits) and `Frame Owner MAC address` (4-bits).

With respect to the receiver-side's arbiter, i.e. Arbiter-L7-R, its flow diagram is

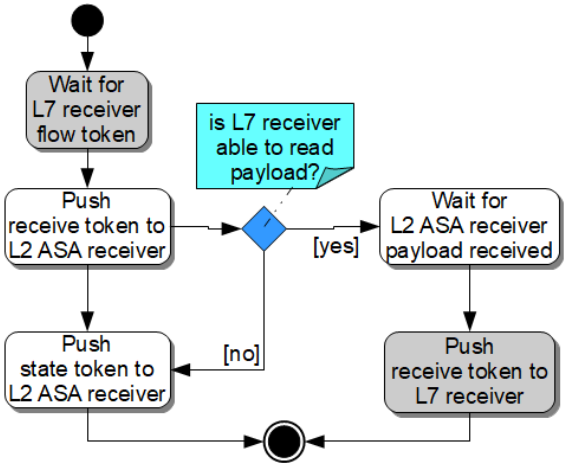


Figure 8-24: Flow diagram of application layer receiver-side arbiter

depicted in figure 8-24. The state information managed within the arbiter, revolve around the reception status of the payload rather than header related states.

L7 L1 arbiter acquisition

Throughout the descriptions provided above, one crucial value is used by and exchanged between all (sub-)layers and their respective sub-components, i.e. frame index. Two layers are responsible for acquiring the frame index within the node design, the receiver-side of the physical layer and the sender-side of the application layer. More specifically, they interact with a dedicated auxiliary that in turn acquires frame indices on their behalf. It is called L7 L1 arbiter acquisition.

The L7 L1 arbiter acquisition auxiliary acts as intermediary between the two layers and all the arbiters within the node design. Within the auxiliary, the acquisition is accomplished by interacting with all arbiters via their dedicated request and response interfaces, i.e. b_put/b_get pairs. Once an arbiter is acquired, the auxiliary pushes the corresponding frame index to the layers via their respective interfaces, which are depicted in figure 8-25.

Unlike the application layer, the physical layer does not make an explicit arbiter request to the auxiliary. As a result, the auxiliary includes only a b_put interface for the physical layer. This is due to the continuous influx of frames in the physical layer, which is in turn is due to the operational nature of the network, i.e. continuously

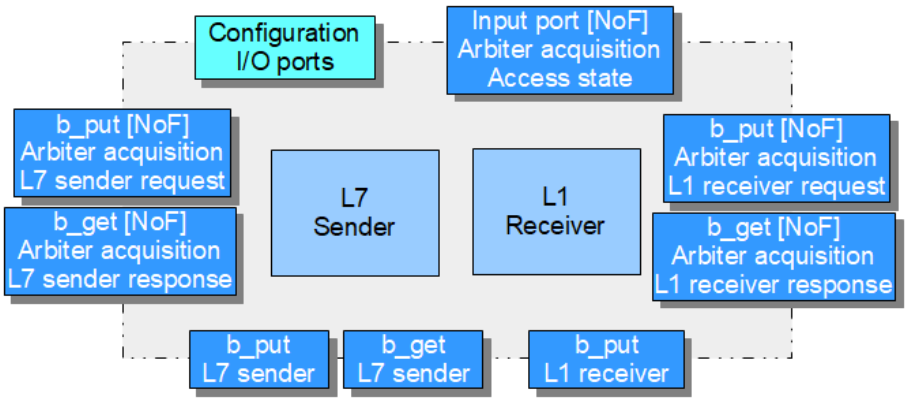


Figure 8-25: L7 L1 arbiter acquisition design

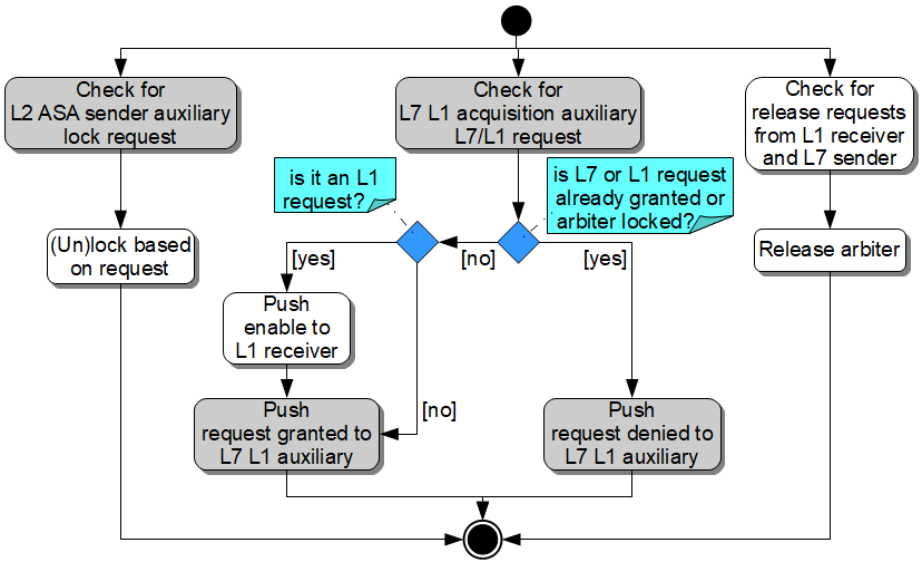


Figure 8-26: Flow diagram of arbiter acquisition sub-component

circulating frames.

On the other end of the auxiliary/arbiters interaction, each arbiter includes a dedicated sub-component that manages and handles the acquisition attempts as well as the aforementioned request and response interfaces. Although this acquisition sub-component does not contain frame/packet state information, it does include acquisition state information. The corresponding flow diagram is depicted in figure 8-26.

8.4.2 Router

Before delving into the router design specifics, in what follows, the reasoning behind implementing and maintaining two separate designs, i.e. node and router, is provided. The main guiding principle is reduced FPGA resource consumption, which in-part leads to reduced power consumption. In other words, more FPGA resources are used only if routing functionality is required. Otherwise, a unified design can be implemented, where it is partially disabled in case of a node. Although maintaining one design is relatively easier, this results in an inefficient use of FPGA resources in case of nodes, which constitute the majority of a network.

This sub-section follows the same structure as the node sub-section above. However, rather than reiterating through mostly similar descriptions as the ones provided above, only noteworthy differences are covered in this sub-section.

As described previously, a router uses both ports included in the prototype board. Within the router design, each port has its own layers, arbiters and memory. Thus, the router design consists of two router port designs. Each router port design has its configuration interface. Therefore, the RouterConfig *SC_MODULE* supplies configuration data that covers both ports of the router. In other words, it includes twice the number of configuration I/Os and reads twice the amount of data from Flash memory. In addition, the configuration data described in the node sub-section above, RouterConfig also retrieves unicast and multicast routing tables.

With respect to the physical layer of a router port design, the sender-side includes an additional memory interface used in the context of routing. Thus, rather than copying data internally, a dedicated interface is used to access the other port's memory.

Within the context of the data-link layer, the ACF sub-layer differs from the node implementation above. Both the sender- and receiver-sides also monitor the clock synchronization status of the other port, thereby achieving across ring synchronization as described in chapter 6. As for L2 ACF SendState, it also takes into account route requests, which are essentially send requests initiated by the receiver-side of the other

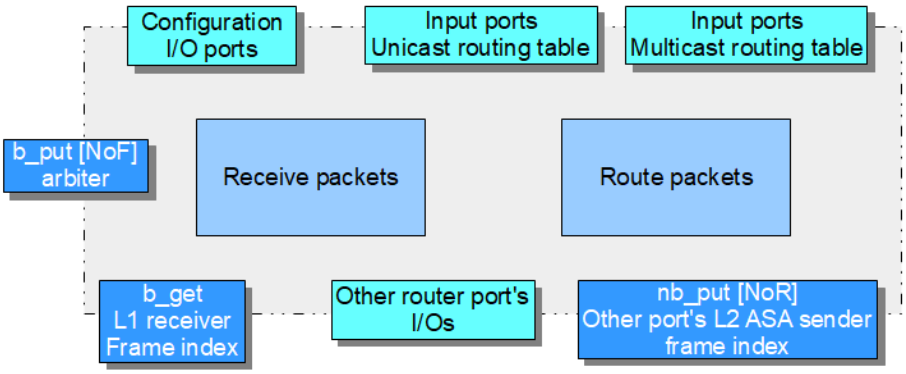


Figure 8-27: Network layer receiver-side design

port's network layer.

While having the same sender-side, the receiver-side of the network layer differs drastically between the node and router designs, as depicted in figure 8-27. Due to the routing functionality, the additional interfaces and I/O ports required, within a router port design, also include some belonging to the other port. More specifically, they include ring initialization status and ring address ports. As for the corresponding arbiter, i.e. Arbiter-L3-R, it has mostly the same flow diagram depicted in figure 8-21. However, the state information pushed to Arbiter-L2-ACF-R also includes route state information.

Chapter 9

Results and Measurements

This chapter presents the results and measurements performed throughout the project lifetime. They are gathered using both Virtex5- and Kintex7-based prototypes. Prior to presenting the results, the first section explains the metrics and scenarios used. Thereafter, the collected measurements are presented based on the prototype used, i.e. in two sections. Finally, the last section covers a lab demonstration in which third-party devices are attached and incorporated into the network.

9.1 Introduction

In order to keep the focus on the results and their interpretations within the subsequent sections, the preliminary descriptions of the results are moved into this section. It provides descriptions pertaining to the metrics and scenarios used below. In other words, this section serves as an introductory as well as a reference for the subsequent sections. Beside metrics and scenarios, this section also describes the pipeline-like effect, which takes place during and has an effect on transmission within the network.

9.1.1 Metrics

Since the protocol is designed and implemented in an FPGA, the time values are expressed in terms of number of clocks, i.e. clock cycles. This more accurately

reflects the protocol behavior. In other words, clock count is independent of the clock period of the available clock signal within the prototype. As described in chapter 2, a design can be switched to a faster clock as long as no negative slack is encountered. Nevertheless, a time value is provided in parentheses next to each clock count.

In what follows, a brief description of each metric used is provided.

Bandwidth

The size(bits) of user data transmitted divided by the transmission time(seconds). In other words, how fast actual user data can be transmitted, i.e. accounts for protocol and transceiver encoding overhead. In relation to Aurora's line rate, this can be considered as the throughput or effective bandwidth.

Delay

The time(seconds) it takes for a frame or packet to completely arrive at the destination, starting from the first bit sent until the last bit received. It is based on timestamps collected from both the sender and receiver, i.e. time(received) - time(sent). Thus, it requires network-wide clock synchronization.

Jitter

It is delay fluctuation or variation measured over a relatively large number of transmissions, i.e. large sample size. The corresponding transmissions have the same source, destination and type. Within the context of CarRing IV, they represent the same user send request.

Jitter can be expressed as a distribution or maximum/minimum bounds. In the case of test scenarios used below, it is the difference between max and min delays measured. Thus, it corresponds to the worst case jitter as opposed to a statistical average or variance of the statistical distribution associated with the delay measurements. As for the number of transmissions, it is 1000.

In CarRing IV, in-order delivery is maintained due to multiple factors: frame/-packet processing approach within node/router designs, shared transmission medium,

i.e. physical ring topology as well as static routes for non-local destinations. Therefore, jitter is not caused by out-of-order delivery within CarRing IV, rather it's due to the minor differences between, when a send request is issued and when an adequate sending opportunity is available.

Elapsed time

It is defined similarly to the delay metric. However, unlike the delay metric, the measurement is entirely contained within either the sender or the receiver depending on the test scenario. In other words, it is not based on timestamps within both sender and receiver. Furthermore, it additionally accounts for returning acknowledges in case of acknowledged transmission.

The measurement is done using a clock counter within the test application, i.e. on top of the application layer. The measurement is collected after 1000 transmissions are completed. The counter's start and stop conditions depend on the test scenario as well as the transmission type, i.e. acknowledged or unacknowledged. Therefore, those conditions are explained alongside the test scenarios below.

This metric is introduced and used in the early phase of the project, i.e. prior to network-wide clock synchronization. It gives insight into the network timing behavior without requiring clock synchronization.

9.1.2 Test scenarios

Before describing each test scenario, in what follows, descriptions that are common to all scenarios are provided.

In all scenarios, the measurements are collected for both operation modes, anisochronous and isochronous. Another common aspect is the number of transmissions conducted. Within each test scenario, the corresponding user send request is repeatedly issued 1000 times.

Two approaches are used to issuing those user send requests, continuous and isolated transmissions. In the continuous approach, user send requests are issued

as fast as possible. The user interface within the application layer is saturated, i.e. there's always a user send request. Thus, the limiting factor is the number of sending opportunities as well as the availability of the next sending opportunity. As for the isolated approach, an artificial gap of one millisecond is introduced between consecutive user send requests. In this case, the limiting factor is the artificially introduced delay, rather than sending opportunities.

These two approaches are used due to the pipelining effect which is described in a dedicated sub-section below. In case of continuous transmission, the pipelines are filled which results in optimal performance. However, in the case of isolated transmissions, the transmissions are interleaved. And consequently the pipelines are not filled. As such, these two cases cover both ends of the spectrum with respect to the pipelining effect.

Scenario 1

In this scenario, there is only one sender which transmits to one receiver. Both belong to the same ring i.e. routing is not required. However, the ring size is varied, starting with a single node where the sender targets itself. As the ring size increases, the sender's local destination changes based on its proximity to the sender. In other words, the receiver is selected as the furthest node from the sender, i.e. it's the sender's predecessor node within the ring.

Both continuous and isolated transmissions are conducted in this scenario. Two frame types are used in those transmissions, `Ethernet Service` and `Acknowledged Point To Point`. The former represents unacknowledged transmissions while the latter represents acknowledged ones.

Since there is only one sender, this scenario exclusively tests the public access frame reuse feature, when anisochronous mode is engaged. The i-to-j feature is not utilized, because it enables frame reuse as the frame is circulating back to its owner or sender, i.e. it involves at least two senders.

With respect to metrics, bandwidth and elapsed time are measured. In case of acknowledged transmission, the clock counter used for elapsed time is contained

within the sender. The start condition is sending the first bit of the first frame. The stop condition is receiving the last acknowledge for the last frame sent.

As for unacknowledged transmissions, the clock counter is contained within the receiver since there are no returning acknowledges. The stop condition is receiving the last bit of the last frame received. As for the starting condition, it is implemented as receiving the first bit of the first frame. Note that there is a minor mismatch between the start condition defined and the one implemented, i.e. sending the first bit versus receiving the first bit. This slight difference is accounted for in the measurements presented in the next section.

Scenario 2

Similar to the previous one, this scenario also uses a single ring with varying size. However, unlike scenario 1, all nodes are simultaneously sending and receiving. Each node transmits to its successor within the ring. Upon receiving a payload, each node identifies the corresponding source MAC address and replies using that address. In this context, replying means issuing a new user send request that targets the sender of the received payload. To avoid infinite replies, custom identifiers are introduced into payloads and used to distinguish between reply- and regular-payloads. The latter triggers a reply.

This test scenario conducts both continuous and isolated transmissions using the `Ethernet Service` frame type. Unlike the previous scenario, the starting ring size is two nodes. This is due to replies being central to this test scenario, i.e. no self-send. When continuous transmission and anisochronous mode are engaged, this scenario exclusively tests the i-to-j frame reuse feature. In this case, public access is not utilized, because there are no idle nodes.

Similar to scenario 1, the metrics measured are bandwidth and elapsed time. Since all nodes are transmitting, a clock counter is included within each node. The starting condition for the clock counter is sending the first bit of the first frame. The stop condition is receiving the last bit of the last reply.

Scenario 3

Similar to scenario 2, all nodes are simultaneously sending and receiving. However, no replies are issued upon receiving a payload. Furthermore, the ring size is kept fixed, while the sender's local destination is varied. The destination MAC address is gradually incremented, where the first destination is the successor node. Due to the ring topology and the way MAC addresses are assigned within the protocol, the last destination increment results in the sender targeting itself.

This scenario conducts continuous transmissions using `Ethernet Service` and `Acknowledge Point To Point` frame types. Similar to scenario 2, it also tests the i-to-j frame reuse feature, when anisochronous mode is engaged. Moreover, the public access feature is also not utilized due to lack of idle nodes. In contrast to the second scenario, this one tests the interferences between medium distance transmissions as opposed to short distance ones in scenario 2.

The measured metrics are bandwidth and elapsed time. Each node contains a clock counter. The counter's start condition is sending the first bit of the first frame. In case of unacknowledged transmission, the stop condition is receiving the last bit of the last frame received. With respect to acknowledged transmissions, the stop condition is receiving the last acknowledge of the last frame sent.

Scenario 4

Unlike all previous scenarios, this scenario involves routing. There is only one sender that transmits to one receiver. The varying parameter is the number of intermediate routers that the packet traverses before reaching its destination. Starting with zero intermediate routers, i.e. the sender transmits packets to a local destination. Adding an intermediate router implies the addition of a new ring into the network. In order to have only one varying parameter, each newly added ring has the same size as the sender's ring. Thus, all rings within the network have the same number of circulating frames, i.e. the same number of sending opportunities.

This scenario conducts continuous transmissions in anisochronous mode. The

`Ethernet Service` and `Acknowledge Point To Point` packet types are used in those transmissions. As for the frame reuse features within anisochronous mode, the public access is exclusively used due to in-part having only one sender, i.e. there are idle nodes/routers. However, more importantly, it is due to the i-to-j feature being explicitly disabled for non-local destinations.

Both the sender and the intermediate routers can use all frames circulating within their respective rings. Thus, this test represents the best case scenario for a sender with non-local destination.

Three metrics are measured in this scenario, bandwidth, delay and jitter. As described previously, sender and receiver timestamps are used for the delay metric. Thus, no clock counters are required within the test application. Within the sender, the timestamp is collected, when the first bit of a packet is sent. As for the receiver, it is collected after receiving the last bit of a packet.

Scenario 5

This scenario can be considered as the counterpart to scenario 4. It represents the worst case scenario for a sender with non-local destination. Unlike the previous scenario, the receiver as well as all intermediate routers additionally send packets to a local destination within their respective rings. Thus, there are no idle nodes/routers within the network. Moreover, this scenario is also considered a stress test for the routing functionality since each router is transmitting both, its own user send requests as well as routed packets. Thus, all sending opportunities available to a router port are used for both, own packets and routed ones.

Similar to scenario 4, the anisochronous mode is engaged, where continuous transmissions are conducted using `Ethernet Service` and `Acknowledge Point To Point` packet types. With respect to frame reuse features, only i-to-j is used, however only for local destinations. In other words, it is neither used by the sender, nor by intermediate routers, when transmitting routed packets.

Last but not least, the same metrics are measured, i.e. bandwidth, delay and jitter. Similarly, no clock counters are required. As for collected sender and receiver

timestamps, the same conditions, as those used in scenario 4, apply.

Scenario 6

This test scenario is essentially scenario 1. The main distinctions between the two scenarios are prototype and protocol features. Scenario 1 was carried out using Virtex5-prototype and without network-wide clock synchronization and multicast. On the other hand, this scenario uses Kintex7-prototype, where the node design implements the aforementioned features. The additional features affect the internal processing time within the node design, thereby affecting how fast the user is able issue send requests, i.e. time interval between successive user send requests.

Due to extenuating circumstances, both within the research group and the project itself, the tests within scenario 6 were limited to continuous transmissions using `Ethernet Service` packet type where anisochronous mode is engaged. In other words, the tests performed did not cover all those conducted in scenario 1. Therefore, both scenarios are included in this chapter and their findings are complementary to one another.

The findings of this test scenario additionally highlight the performance gains achieved after migrating CarRing IV designs from Virtex5- to Kintex7-prototype.

9.1.3 Pipeline effect

This effect was briefly described in sub-section 5.1.4 and 8.4.1. This sub-section elaborates more on the pipeline effect, especially how it affects the node transmission behavior and consequently the network's performance.

At the physical layer and between two consecutive nodes, a frame is pushed to Aurora's design component followed by the underlying GTX transceiver. After traversing the network cable, it goes through the GTX transceiver followed by the receiver's Aurora. The latency associated with this frame path is due to the transceivers as well as the data paths within Aurora's design sub-components. As long as those internal data paths are not blocked, i.e. the current frame is being successfully transmitted,

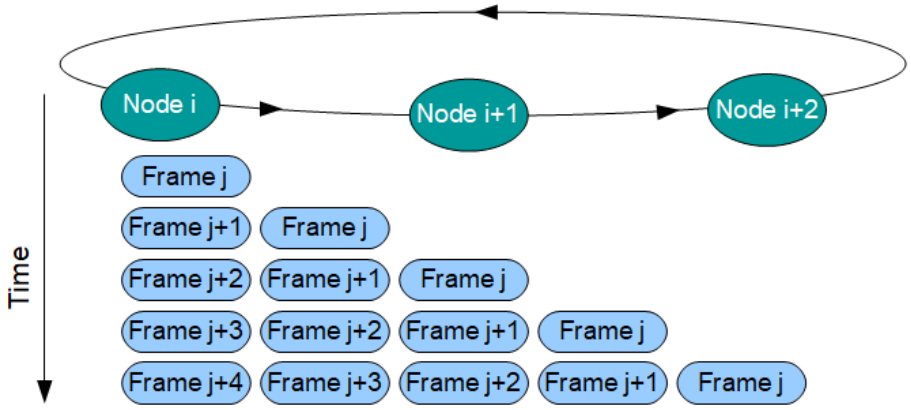


Figure 9-1: Illustration of the pipeline effect

Aurora's user interface simultaneously accepts new frames for transmission.

In order to take advantage of Aurora's behavior, inter-frame gap must be reduced to zero, thereby filling the link between two consecutive nodes. This transmission behavior compounded with the physical ring topology of the network, i.e. multiple intermediate nodes(transceivers) and links, results in a pipeline-like effect, which is illustrated in figure 9-1.

In the illustration, only one sender is transmitting within the ring, where node(i) is targeting node(i+2). In anisochronous mode, node(i) can reuse all frames circulating within ring via the public access feature. Consequently, as frame(j) is being transmitted through intermediate links and nodes, node(i) is simultaneously sending new payloads using its own frames as well as frames belonging to other nodes.

This effect directly impacts the network's performance. More specifically, it affects the maximum achievable bandwidth and the total transmission time of large amounts of user data, i.e. data that requires multiple packets/frames to be transmitted. Thus, bandwidth calculation is not purely dependent on the delay metric.

9.2 Virtex5-prototype

In what follows, the test-bench and conditions common to all test scenarios below are described. Although those tests are conducted in the early phase of the project, they give an important insight into the network and protocol behavior and are therefore

included.

Due to production limitations, only five Virtex5-prototypes are used in the tests below. They are interconnected using one meter shielded twisted copper cables. Within the prototype, each port has one lane and operates at a line rate of 3.125 Gbps. As previously described, the frame size is fixed to 16 aurora-words. With four bytes per lane, this translates to 512-bits frame size. Last but not least, CRC is not delegated to Aurora in the early phase of the project. As such, it consumes one out of the 16 aurora-words used for the frame.

9.2.1 Scenario 1

This scenario is conducted using a test application that connects directly to the data-link layer. In other words, it is a data-link layer test, where the application layer resides directly above data-link. Thus, the frame's 16 aurora-words or 512-bits are allocated as follows: 32-bits for header, 448-bits for payload and 32-bits for CRC.

In all subsequent figures, measurements are depicted using four bars. Each bar corresponds to a combination of an operation mode and a transmission approach. The first two bars correspond to continuous transmission, where anisochronous and isochronous modes are engaged, respectively. As for the remaining two bars, they correspond to isolated transmission with anisochronous and isochronous modes, respectively.

Within the context of unacknowledged transmission, figure 9-2 presents the measured bandwidth for all operation mode and transmission approach combinations. In case of anisochronous mode and continuous transmission, the sender reaches a maximum bandwidth of 1.84 Gbps. Regardless of the ring size, the bandwidth does not degrade.

As the ring size increases, each added node introduces its own three frames into the ring. This results in an increased number of sending opportunities, when anisochronous mode is engaged. Combined with public access frame reuse feature, the sender is able to use all frames circulating within the ring. Consequently, the maximum bandwidth of 1.84 Gbps is maintained regardless of the ring size. Despite

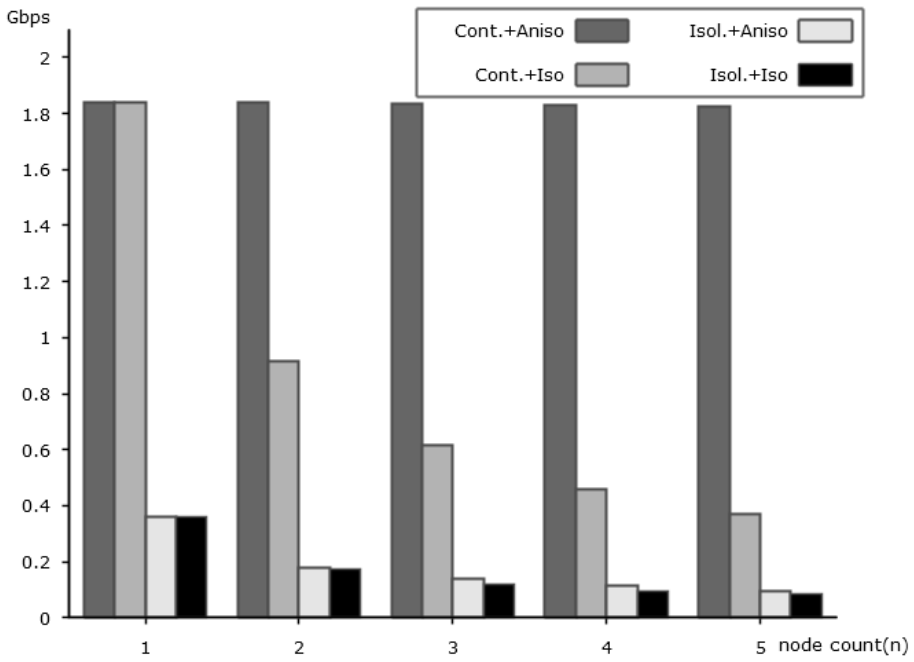


Figure 9-2: Scenario 1 bandwidth measurements for `Ethernet Service`

being limited to five prototypes/nodes, this observation can be extended to 16 nodes, i.e. the maximum ring size. Within the context of the aforementioned combination, block user data transfer is not affected by the ring size.

As for the other three combinations, the bandwidth degrades within isochronous mode is engaged and/or isolated transmission is used. There are fewer sending opportunities in isochronous mode. Whereas, isolated transmission essentially removes the pipeline effect described above.

The elapsed time measurements presented in 9-3 are complementary to the bandwidth ones presented above. The single node ring test essentially represents a loopback test. And the corresponding measurements represents the average elapsed time associated with sending and receiving a frame. It is 19 clocks (243.2 ns) for continuous transmissions. Similar to the previous observation, the combination of anisochronous mode and continuous transmission results in a constant elapsed time, i.e. unaffected by the ring size.

On the other hand, the measured bandwidth and elapsed time for acknowledged transmissions are presented in figures 9-4 and 9-5, respectively. As the ring size

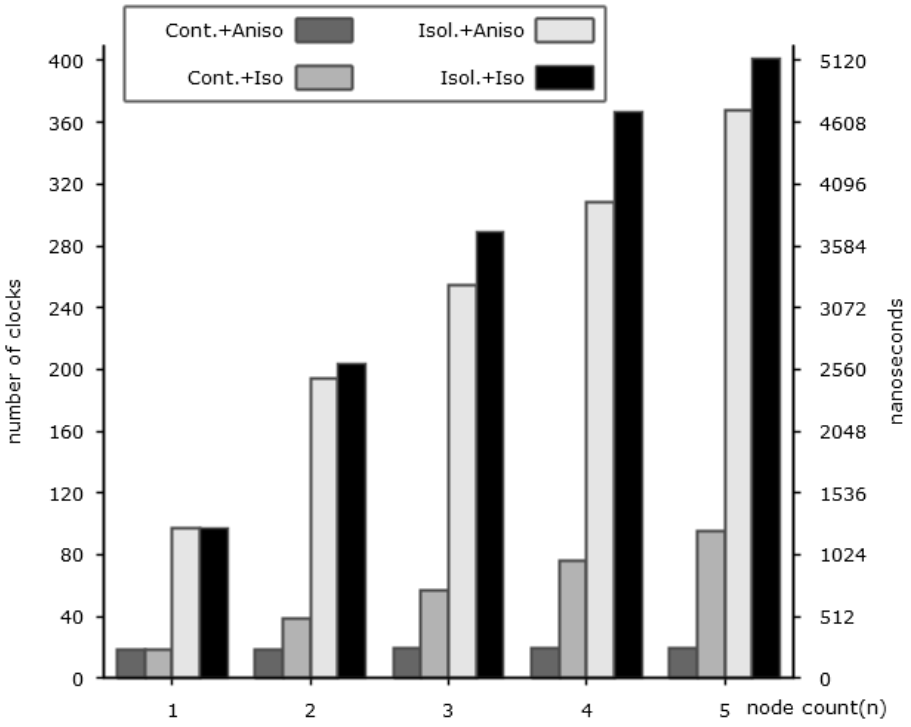


Figure 9-3: Scenario 1 elapsed time measurements for `Ethernet Service`

increases, the performance degrades with lower bandwidth compared to unacknowledged transmission counterparts. This is an expected behavior for the following three combinations, isochronous/continuous transmissions and both of isolated transmission variants. It is due to acknowledgments, which offer higher reliability at the expense of performance.

As for the fourth combination, i.e. anisochronous/continuous transmission, the degraded performance is not caused by acknowledgments, rather limited FPGA resources. Using the public access frame reuse feature, user send requests with acknowledged frame types have access to the same number of sending opportunities as unacknowledged ones. Combined with immediate acknowledges, i.e. using the same frame for the returning acknowledge, the measured bandwidth 0.83 Gbps and elapsed time 42 clocks (537.6 ns) for the fourth combination does not match the expected result. Similar observations as well as comparable bandwidth to the unacknowledged transmissions is expected.

This is caused by FPGA resource limitations, rather than protocol or design imple-

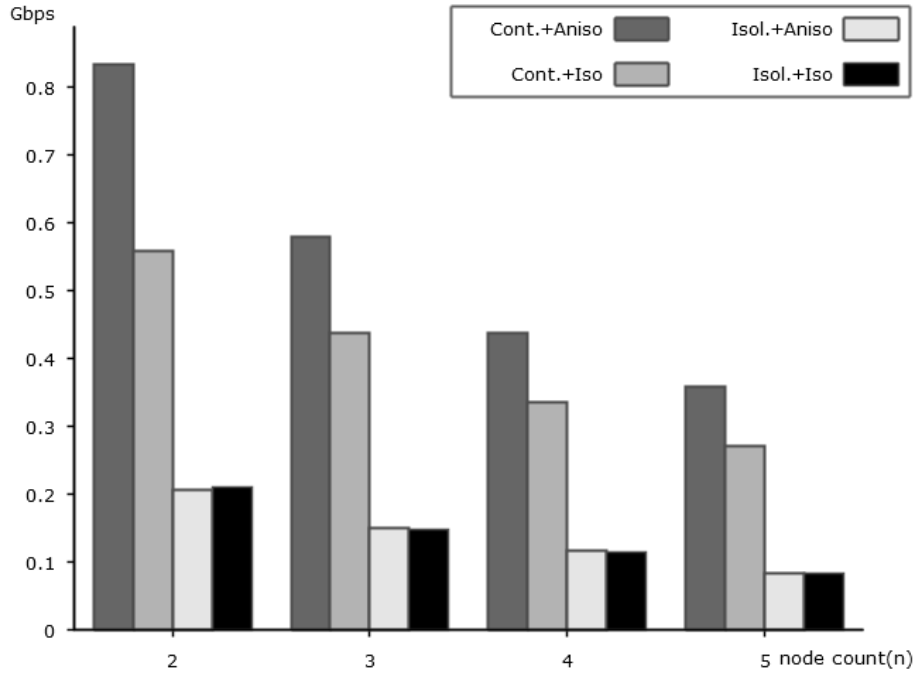


Figure 9-4: Scenario 1 bandwidth measurements for `Acknowledged Point To Point`

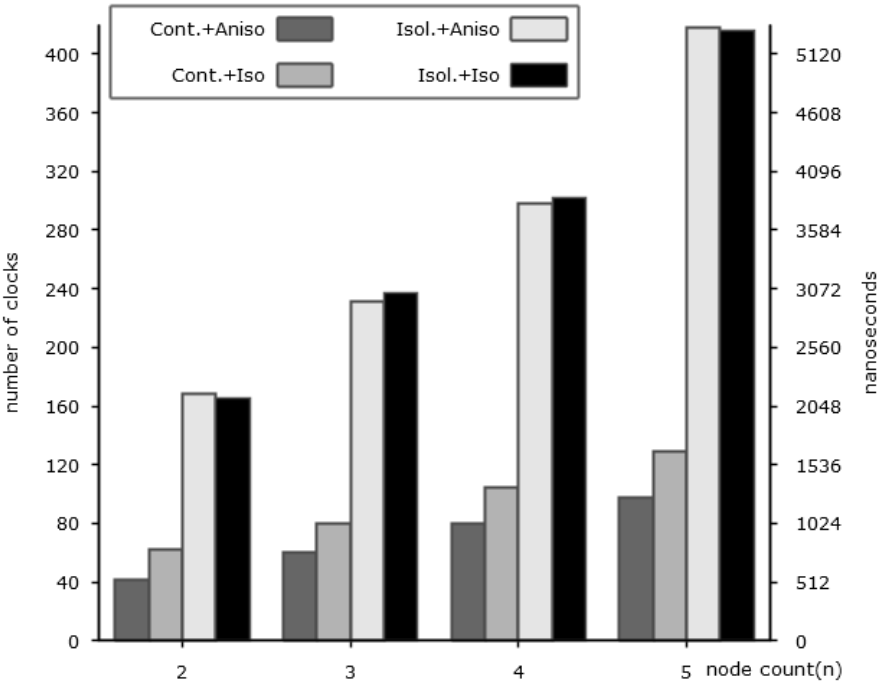


Figure 9-5: Scenario 1 elapsed time measurements for `Acknowledged Point To Point`

mentation. Due to possible retransmissions and time-out events, acknowledged transmissions require additional resources to track them internally within the node/router designs. In order to adapt to the available FPGA resources, the number of permitted ongoing acknowledged transmissions is less than the number of sending opportunities available within the ring. As a result, not all available sending opportunities are used.

9.2.2 Scenario 2

Similar to scenario 1, the same number of protocol layers and frame format are used. Furthermore, the measurements presented in figures below are structured using the same four-bars scheme, i.e. using the same number of operation mode and transmission approach combinations. However, unlike scenario 1, only unacknowledged transmissions are conducted.

Regardless of the combination, bandwidth is directly impacted by ring size, as shown in figure 9-6. Even in the case of anisochronous mode and continuous transmission, performance degrades as ring size increases. Despite lower bandwidth, the combination of anisochronous and continuous transmission still outperforms the isochronous counterpart. This is due to the increased number of sending opportunities via the i-to-j frame reuse feature.

On the other hand, the combination of anisochronous and isolated transmission is not impacted by ring size, where the bandwidth remains at 0.31 Gbps. This consistent behavior is due to in-part the public access frame reuse feature. Whenever a node has no pending user send requests, it enables frame reuse for its own `Transport` frame, i.e. sets the `Public access` header field to 1. Thus, after one millisecond of no send requests, all circulating frames can be used by any node within the ring, thereby a sending opportunity is directly available.

As shown in figure 9-7, similar observations can be made for the elapsed time measured. With respect to the combination of anisochronous and isolated transmission, the elapsed time measured remains at approximately 114 clocks (1452.9 ns), i.e. unaffected by the ring size.

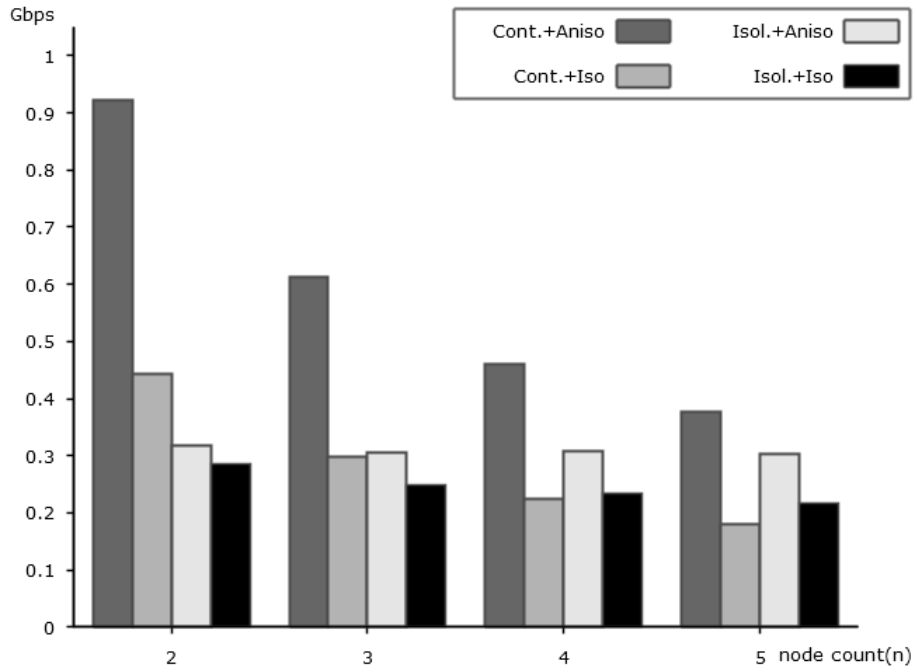


Figure 9-6: Scenario 2 bandwidth measurements for `Ethernet Service`

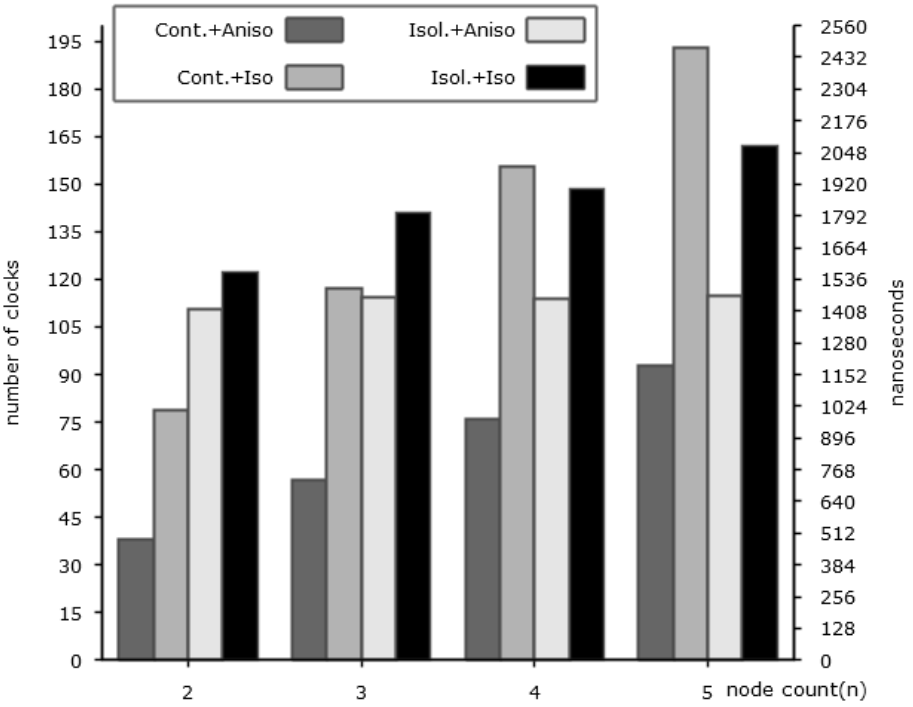


Figure 9-7: Scenario 2 elapsed time measurements for `Ethernet Service`

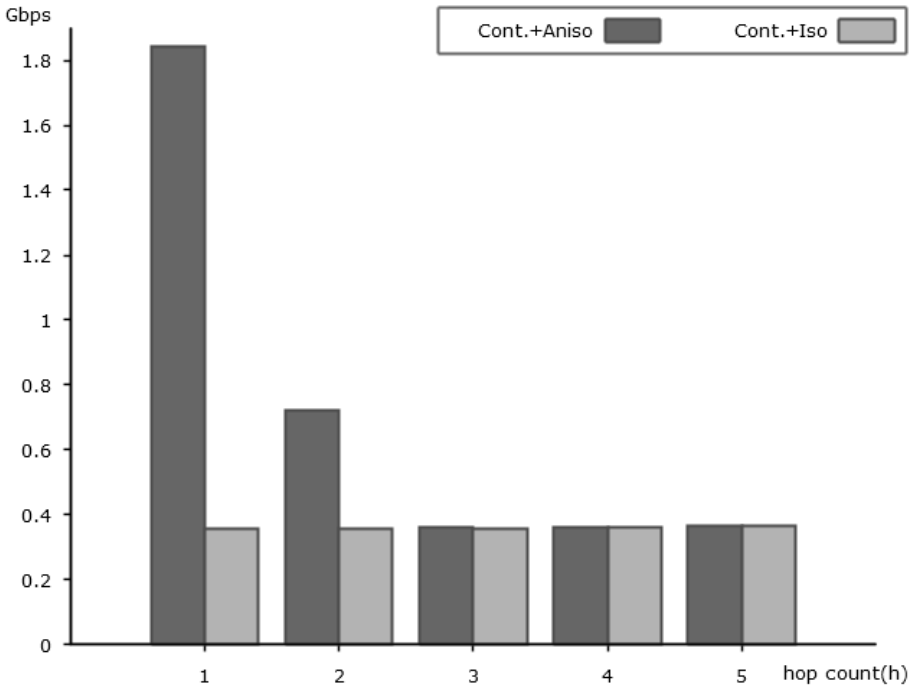


Figure 9-8: Scenario 3 bandwidth measurements for `Ethernet Service`

9.2.3 Scenario 3

This scenario uses the same number of protocol layers and frame format as those described in scenarios 1 and 2. However, unlike the previous scenarios, only two operation mode and transmission approach combinations are used. More specifically, the continuous transmission is used and combined with both operation mode variants . Therefore, only two bars are used in the figures, anisochronous/continuous followed by isochronous/continuous. With respect to the test application, both `Ethernet Service` and `Acknowledge Point To Point` frame types are used, thereby representing unacknowledged and acknowledged transmissions.

As shown in figure 9-8, anisochronous and continuous transmission remains the highest performing combination, i.e. similar to the findings of the previous two scenarios. Furthermore, in case of unacknowledged transmission, the same bandwidth of 1.84 Gbps is reached as in scenario 1. This is due to the i-to-j frame reuse feature. As for the public access feature, it is essentially excluded, because there are no idle nodes within the ring.

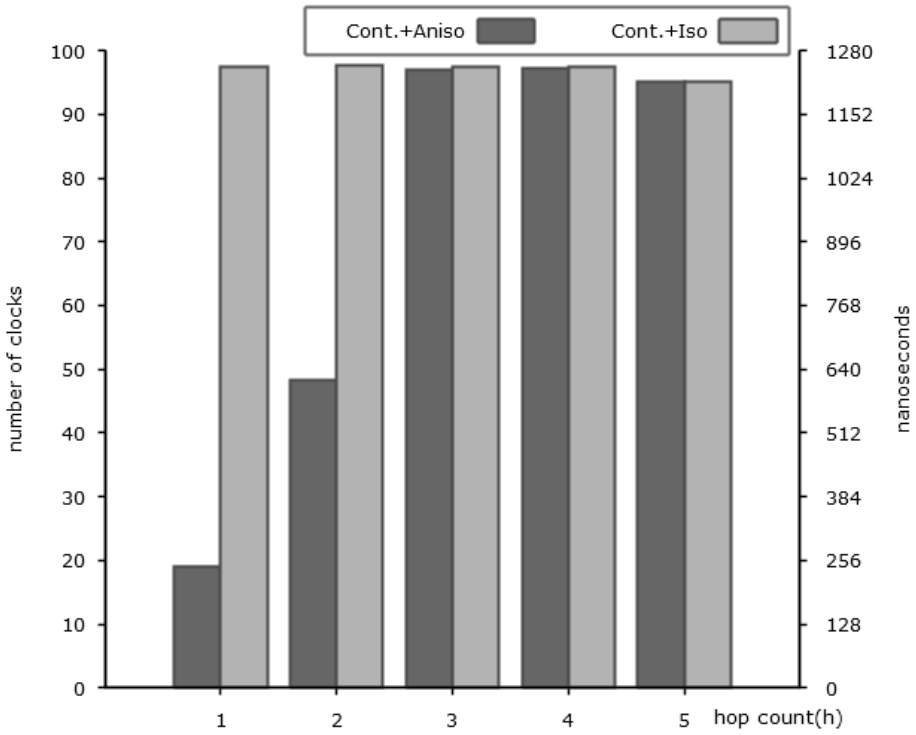


Figure 9-9: Scenario 3 elapsed time measurements for `Ethernet Service`

Nevertheless, the bandwidth degrades as the number of intermediate nodes or hops between a sender and receiver increases. As described previously, the i-to-j feature enables frame reuse based on the positions of the sender and it's target relative to the frame's owner node. In short, a frame can be reused on its way back to its owner node. Thus, the shorter the return path is the fewer possibilities for the frame to be reused. This is reflected within the bandwidth and elapsed time measurements in figures 9-8 and 9-9. The i-to-j feature is used only with hop count 1 and to a lesser extent with hop count 2.

Due to lack of i-to-j feature, the combination of isochronous and continuous transmission is unaffected by increased hop count between sender and receiver. As shown in figures 9-8 and 9-9, it reaches a bandwidth of 0.36 Gbps with an elapsed time of 97 clocks (1241.6 ns).

On the other hand, the acknowledged transmission measurements are presented in figures 9-10 and 9-11. In this case, the i-to-j feature is excluded in both operation modes. Since only acknowledged frame types are used, senders can not utilize the

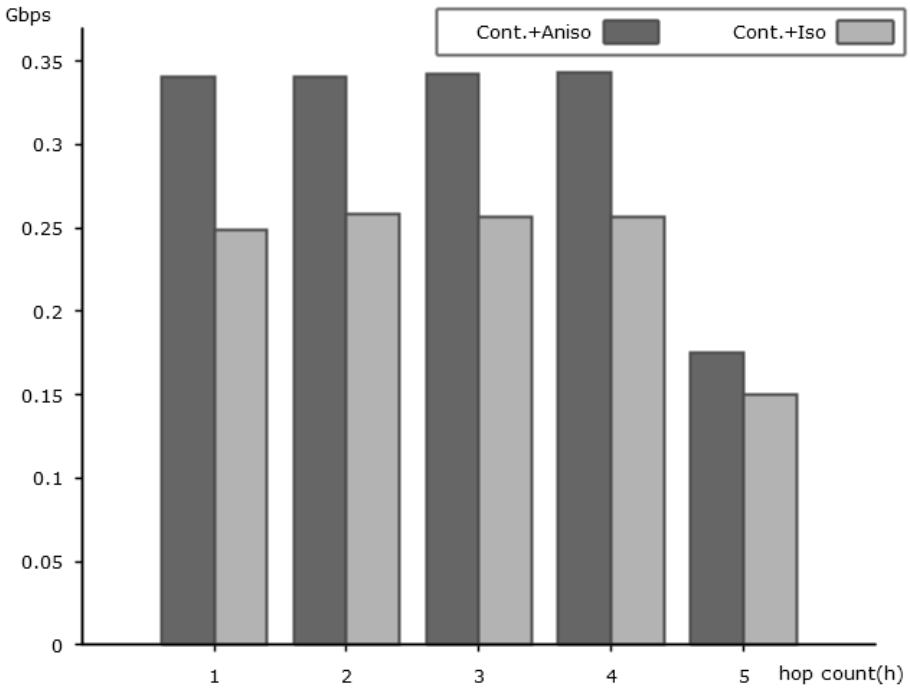


Figure 9-10: Scenario 3 bandwidth measurements for `Acknowledged Point To Point`

i-to-j feature. This is due to immediate acknowledges, where the same frame carries an acknowledge on its way back from the receiver to the original sender. In short, i-to-j can be used with unacknowledged transmissions. This includes frames carrying routed packets that in turn use an acknowledged type, i.e. acknowledges are issued for local destinations only.

When the hop count reaches five, i.e. ring size, each sender targets itself. As a result, a frame associated with an acknowledged transmission traverses the ring twice. Once to deliver user data and then again for the corresponding acknowledge. As shown in figures 9-10 and 9-11, bandwidth and elapsed time are unaffected by the combination used, as long as the hop count is smaller than the ring size. Thus, senders are not targeting themselves, i.e. the corresponding frame traverses the ring once.

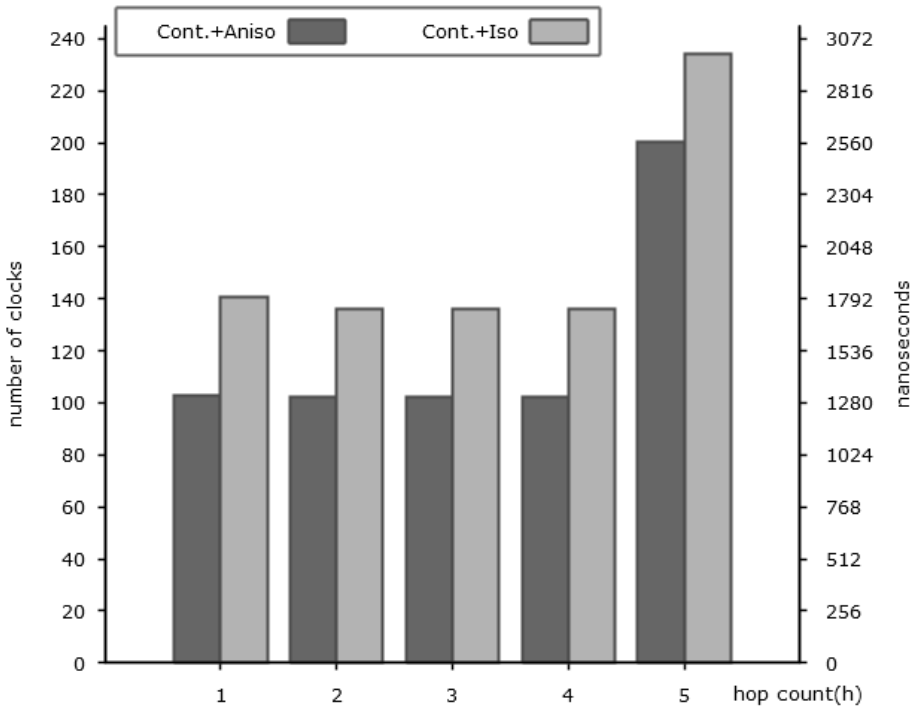


Figure 9-11: Scenario 3 elapsed time measurements for `Acknowledged Point To Point`

9.2.4 Scenario 4

In this scenario, the application layer resides on top of the network layer. Therefore, it is a complete test that includes all protocol layers. Thus, the frame/packet format is as follows: 32-bits for the frame's header, 16-bits for the packet's header, 432-bits for payload and 32-bits for CRC. Using the five Virtex5-prototypes, a total of four rings are created, where each ring has a size of two nodes. This is possible due to routers having two node addresses. Essentially, they operate as two nodes belonging to two rings.

Since delay is used instead of the elapsed time metric, this scenario relies on network-wide clock synchronization. The reference node is chosen based on its proximity to all other nodes within the network, i.e. situated as close as possible to the middle of the network. In case of one intermediate router, the router itself is chosen to be the primary for both rings as well as the reference node. With two intermediate routers, the first router is the reference node, where each port is a primary for the

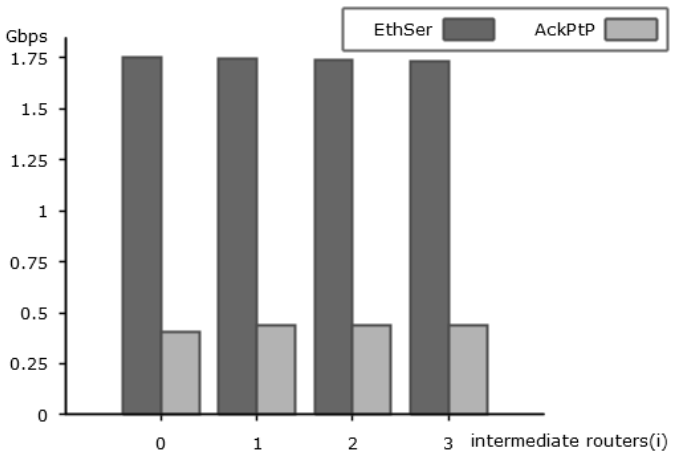


Figure 9-12: Scenario 4 bandwidth measurements

corresponding ring. Finally, with three intermediate routers, the second router is the reference node as well as primary for its rings.

This scenario also uses a two-bars scheme to present measurements within figures. However, each bar stands for a packet type as opposed to a combination of operation mode and transmission approach. The latter two are fixed to anisochronous mode and continuous transmission. As for the packet types, the first bar depicts `Ethernet Service` and the second bar depicts `Acknowledge Point To Point` measurements. The packet types represent unacknowledged and acknowledged transmissions, respectively.

As shown in figure 9-12, bandwidth is unaffected by the number of intermediate routers between the sender and receiver. It remains at approximately 1.75 Gbps for `Ethernet Service` and 0.4 Gbps for `Acknowledge Point To Point`. Similar to scenario 1 observations, this consistent behavior is due to the public access feature as well as the pipeline effect described in sub-section 9.1.3. Also, the performance gap between unacknowledged and acknowledged transmissions is due to fewer number of sending opportunities utilized. As described in scenario 1, this originates from an FPGA resource limitation as opposed to protocol or implementation.

Since acknowledgments are only issued for local destinations, the delay metric is measured using the acknowledge received timestamp for i=0, whereas the payload received timestamp is used for i≥1. As expected, delay is directly affected by the

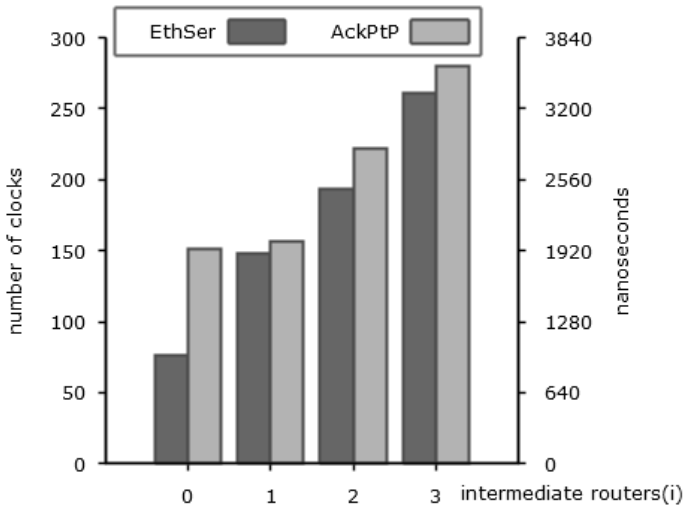


Figure 9-13: Scenario 4 delay measurements

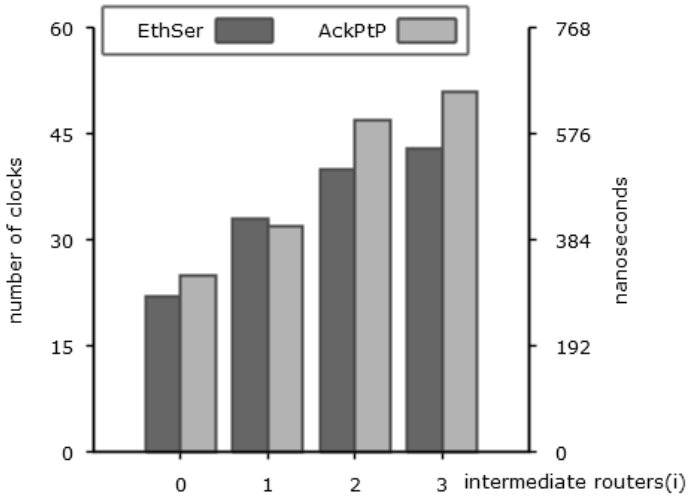


Figure 9-14: Scenario 4 jitter measurements

number of intermediate routers. It ranges from 76 clocks (972.8 ns) to 261 clocks (3340.8 ns) for `Ethernet Service`, and from 152 clocks (1945.6 ns) to 280 clocks (3584 ns) for `Acknowledged Point To Point`. As shown in figure 9-13, the difference between unacknowledged and acknowledged transmissions diminishes as the number of intermediate routers increases.

When compared to scenario 1, the elapsed time for `Ethernet Service` is slightly increased from 19 clocks (243.2 ns) to 20 clocks (256 ns). This is due to the added functionality, i.e. the network layer. As for jitter, the measurements are presented in figure 9-14. Similar to delay, it also linearly increases with the number of intermediate

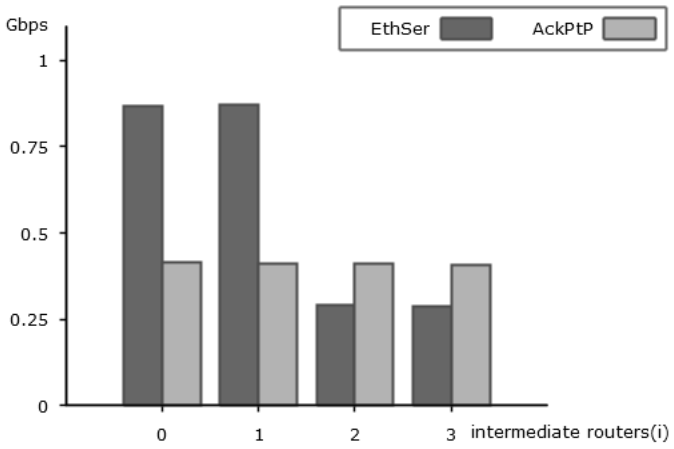


Figure 9-15: Scenario 5 bandwidth measurements

routers. It starts with 22 clocks (281.6 ns) up to 43 clocks (550.4 ns) for `Ethernet Service` and 25 clocks (320 ns) up to 51 clocks (652.8 ns) for `Acknowledged Point To Point`. The jitter is very small which is ideal for real-time applications.

9.2.5 Scenario 5

As described previously, this scenario is considered both the counterpart of scenario 4 as well as a stress test for the routing functionality. As such, the number of protocol layers and frame format are identical to those of scenario 4. Furthermore, the same network topology is used with respect to the size of the rings and the location of the reference node. As for the measurements presented in figures below, the same packet types and two-bars scheme are used.

The receiver as well as all intermediate routers are processing user send requests and transmitting user data as fast as possible to local destinations. As expected, the bandwidth degrades due to reduced number of sending opportunities for both, the sender as well as intermediate routers. When compared with scenario 4, the bandwidth for unacknowledged transmissions drops from 1.75 Gbps to 0.87 Gbps for i=[0,1] and to 0.29 Gbps for i=[2,3] as shown in figure 9-15.

As described previously, user send requests are issued as fast as possible within intermediate routers. Despite this continuous input from the application layer, packets are simultaneously routed. In other words, bandwidth never drops to zero. This

is due to frame and packet processing approach within the router design, i.e. it is based on time of arrival and no prioritization of any form is used.

In scenario 4, the sender transmits packets as fast as possible, which translates to a packet every 20 clocks (256 ns) regardless of the number of intermediate routers. In contrast, the scenario 5 sender transmits a packet every 39 clocks (499.2 ns) for i=[0,1], every 115 clocks (1472 ns) for i=2 and every 117 clocks (1497.6) for i=3. While a lower sending rate is expected, it is unaffected if the non-local destination is in an adjacent ring. Meanwhile, it slowly degrades for i>1.

As for delay, it exhibits a nearly linear increase, ranging from 116 clocks (1484.8 ns) to 285 clocks (3648 ns) for `Ethernet Service` and from 149 clocks (1907.2 ns) to 325 clocks (4160 ns) for `Acknowledged Point To Point`. This is comparable to the delays measured in scenario 4. Thus, network traffic within intermediate rings has minimal impact on delay.

On the other hand, jitter is heavily impacted by increased traffic within intermediate rings. As shown in figure 9-16, in case of i=3, it reaches up to 117 clocks (1497.6 ns) for `Ethernet Service` and 137 clocks (1753.6 ns) for `Acknowledged Point To Point`. It is approximately 2.7 times higher than scenario 4 jitter. Therefore, very low jitter can not be guaranteed if the user does not consider and adequately manage network resources. This is not considered a limitation, because tasks/events within real-time systems are well-defined and managed.

9.3 Kintex7-prototype

Similar to section 9.2, in what follows, the testbench and conditions used in scenario six below are described. As mentioned in sub-section 9.1.2, extenuating circumstances limited the number of tests conducted using the Kintex7-prototype. Among those circumstances, expired license to EDA tools and limited power supply equipment. As a result, only one test scenario is conducted in which only ten Kintex7-prototypes are used.

The prototypes are interconnected using one meter Twinaxial cables. Unlike the

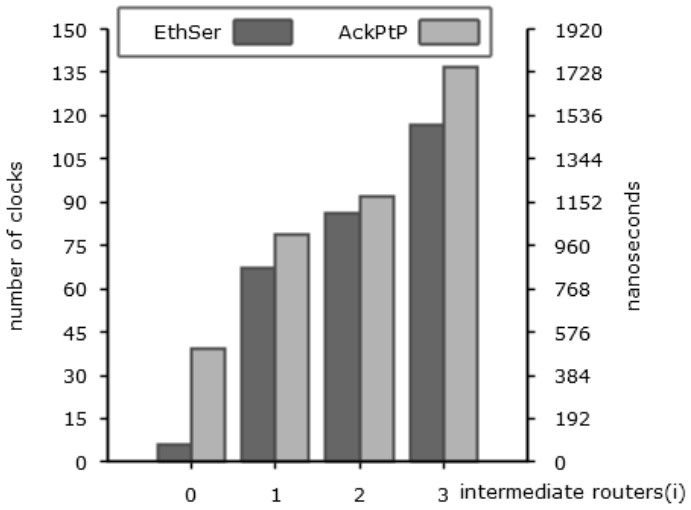


Figure 9-16: Scenario 5 jitter measurements

previous prototype, Kintex7-prototype has a total of eight lanes. As such, each port has four lanes and operates at a line rate of 6.25 Gbps. The latter is due to in-part improved network cables. As for frame and packet formats, they are exactly as described in chapter 5, i.e. 24-bits frame header followed by 16-bits packet header and a 2008-bits payload. With four bytes per lane, the frame size remains at 16 aurora-words, i.e. 16 x 128-bits.

9.3.1 Scenario 6

All previous scenarios gave insight into the network and protocol behavior. This scenario is aimed more at the performance of the network and protocol. As such, the measured bandwidth and elapsed time represent CarRing IV capabilities.

In this scenario, all protocol layers are used, i.e. it's a complete test. As described previously, the test is conducted using only one combination, which is anisochronous and continuous transmission. Therefore, the measurements are depicted using one bar within figures below. Since only `Ethernet Service` packet type is used, the measurements below represent unacknowledged transmissions.

As shown in figure 9-17, the sender reaches a bandwidth of approximately 13.61 Gbps. Similar to the observation in scenario 1, the bandwidth does not degrade as ring size increases. The sender is able to transmit a packet every 23 clocks (147.2 ns)

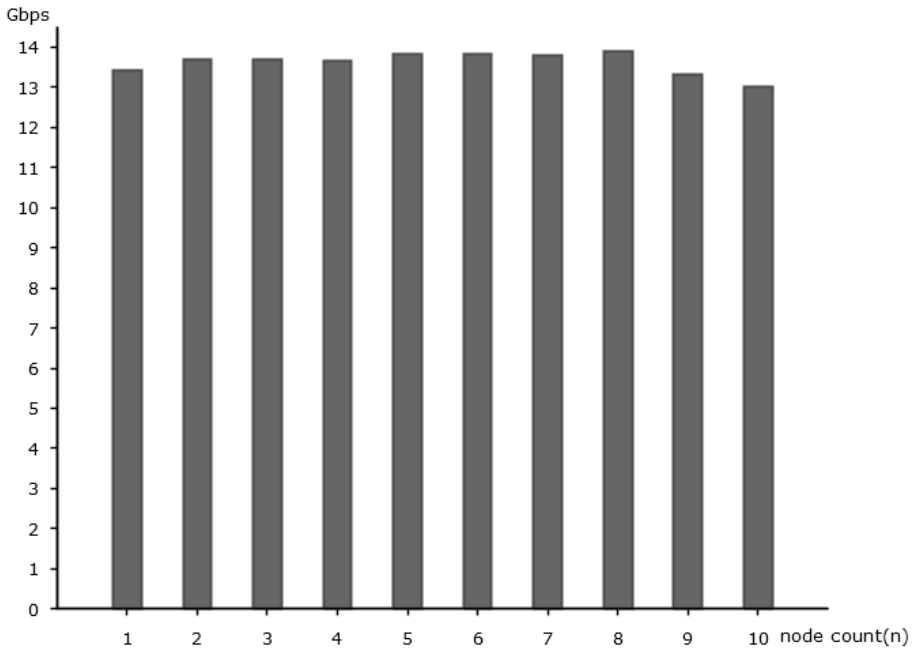


Figure 9-17: Scenario 6 bandwidth measurements

compared to 19 clocks (243.2 ns) for scenario 1 and 20 clocks (256 ns) for scenario 4.

As described in the application layer portion of sub-section 8.4.1, there are no internal checks on time intervals provided by a user application within the context of send-at requests. The time interval of 23 clocks (147.2 ns) can be used as a lower limit, when issuing send-at requests.

Similarly, elapsed time is unaffected by the ring size as shown in figure 9-18. The protocol and its implementation underwent major changes, since the tests conducted in scenario 1. These changes include added functionality and features. Furthermore, migrating from Virtex5-prototype to Kintex7-prototype also resulted in implementation specific changes to accommodate the new clock domain, i.e. from 12.8 ns to a 6.4 ns clock domain. Despite those changes, scenario 1 findings still holds. Moreover, the extrapolation, with respect to ring size effect, made in scenario 1 is confirmed by the test results of this scenario.

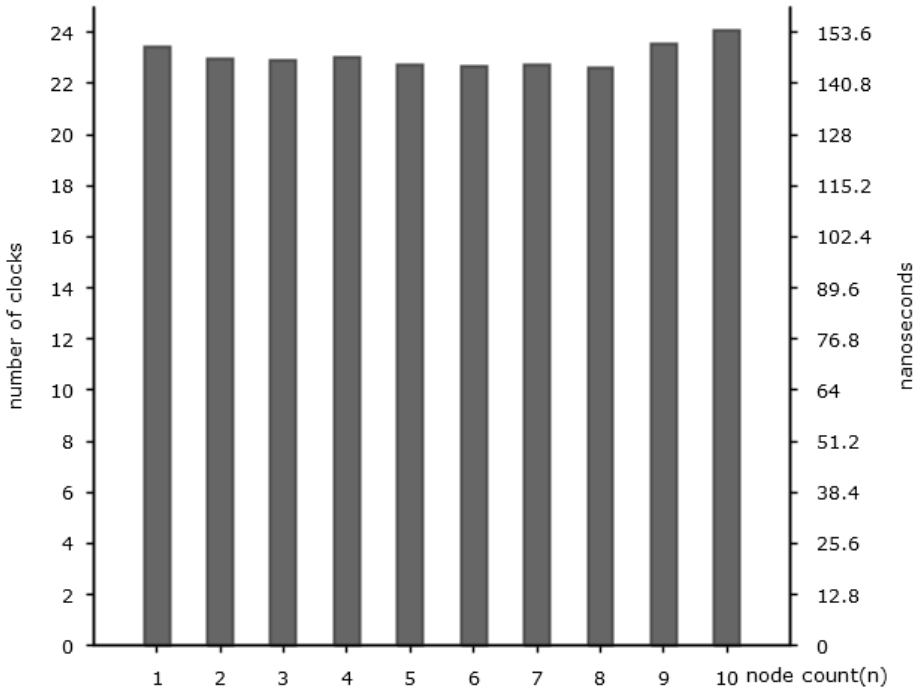


Figure 9-18: Scenario 6 elapsed time measurements

9.4 Tech demo

As briefly described in section 4.3, a lab demonstration is carried out using the five Virtex5-prototypes. This is part of the results achieved in the project and serves as a proof of concept. In what follows, the demonstration as well as the attached devices and their respective configurations are presented.

Similar to the test scenarios in section 9.2, the Virtex5-prototypes are interconnected using one meter shielded twisted copper cables. And their respective ports are configured to operate at a line rate of 3.125 Gbps. The network consists of two rings with three nodes each. Those rings are interconnected by one router. Thus, one prototype implements the router design, while the others implement the node design. All protocol layers are used, i.e. it's a complete test. As for the frame/packet format, the 16 aurora-words are structured as follows: 32-bits frame header, 16-bits packet header, 432-bits payload and a 32-bits CRC.

As shown in figure 9-19, the attached devices include:

- A Laptop streaming a video. It's connected via the Ethernet interface to

node(2.0), i.e. node with ring address 2 and MAC address 0. In turn, node(2.0) communicates with node(1.0), which connects to the Internet Service Provider (ISP) via its Ethernet interface. Both Ethernet connections operate at 100 Mbps.

- IP camera connected to node(1.1) using the Ethernet interface. Its counterpart is a Laptop that hosts the camera's control software, which displays the live feed. The Laptop is connected to node(2.1) via the Ethernet interface. Similarly, the Ethernet connections operate at 100 Mbps.

- Two motors are introduced into the network. They are distinguished based on the interface used to control them. Using the CAN interface, the CAN motor is connected to node(1.0). Its control software is hosted by another Laptop, which in turn is connected to node(2.0) via its CAN interface. Both CAN connection operate at 125 kbps.

- The Laptop that hosts the control software and the corresponding RS232 motor are connected via the RS232 interface to nodes (1.0) and (2.0), respectively. Their connections operate at 9600 baud rate.

- Last but not least, the car headlight is connected to an intermediate custom board that in turn connects to node(1.1) via the IIC interface. On the other side of the network, the car headlight is controlled via the test application of node(2.1). The IIC connection operates in standard mode at 100 kHz.

As described in sub-section 5.4.6, the application layer includes a feature called, transparent mode. Within this demo, it is used to transparently transfer data between multiple supported physical interfaces, i.e. interfaces that are part of the prototype. The transparent mode is implemented as a multi-clock domain design that consists of multiple design components aimed at configuring and controlling the used physical interfaces. Figure 9-20 depicts the design components, clock domains and chips used in the design.

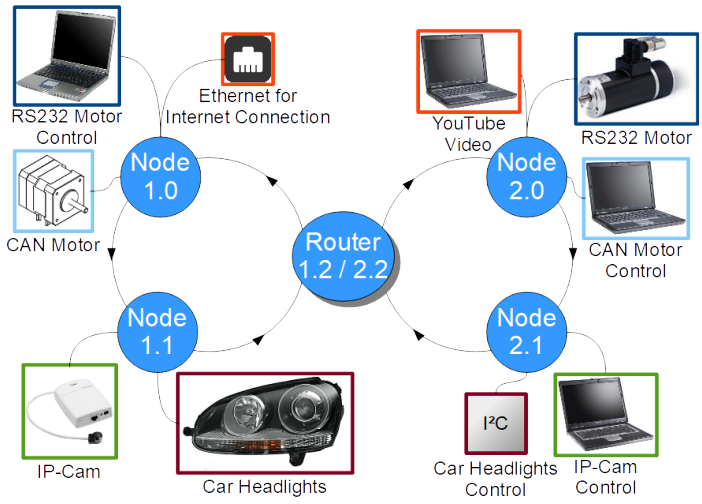


Figure 9-19: Lab demonstration of CarRing IV

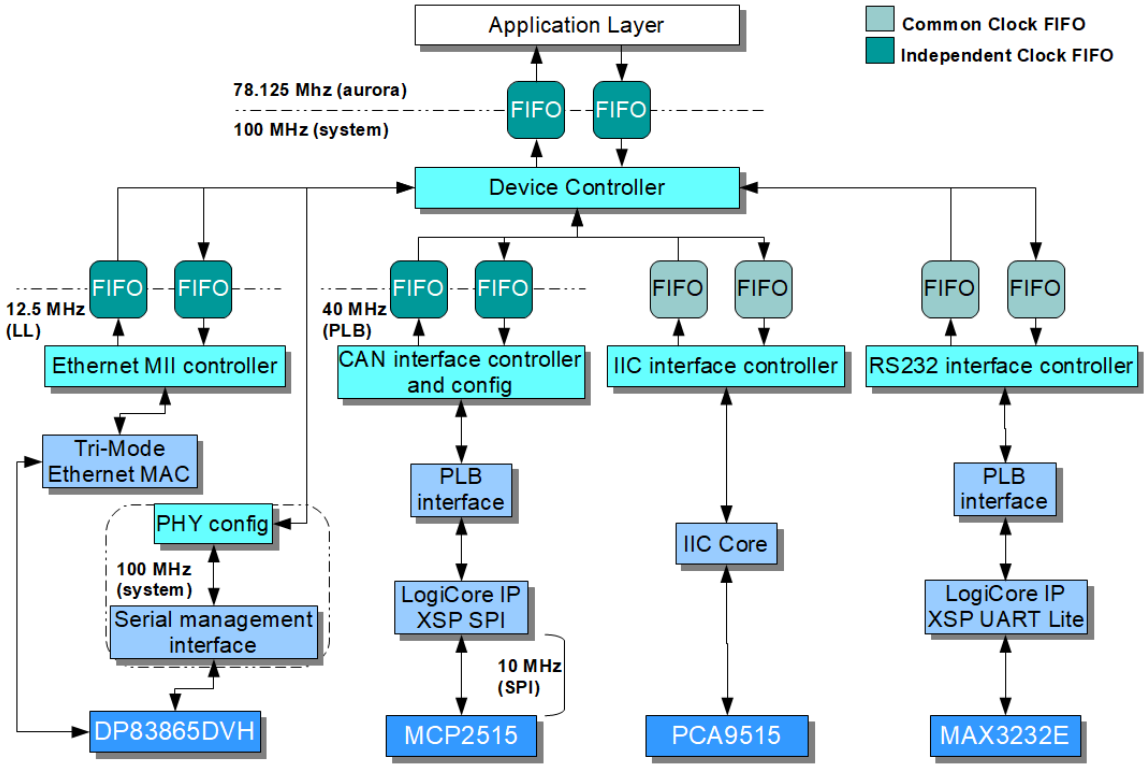


Figure 9-20: Device controller design used in lab demo

The purpose and function of the device controller as well as the interface specific controllers are described in sub-sections 5.4.7 and 5.4.8. With respect to Ethernet, the transceiver is configured to operate at 100 Mbps prior to data transmission. This is due to performance and FPGA resource limitations. As such, additional design components are created and introduced to communicate with the transceiver, i.e. PHY config and SMI.

In the case of the CAN interface, the underlying transceiver uses SPI as the interface through which the corresponding chip can be configured and used. Unlike Ethernet's SMI, a Xilinx SPI IP core is available however only for embedded systems. Therefore, a Xilinx Embedded design is created, where no hard or soft processors are included. Only the SPI IP core is included, where its I/O ports are exposed, i.e. configured as external ports of the embedded design. Since it is meant for embedded systems, an additional design component is created and introduced to communicate with the IP core i.e. PLB interface.

Using the same approach, the RS232 interface controller uses the PLB interface to control the XPS UART core, which in turn controls the underlying chip. As for IIC, a custom design is created to communicate with the corresponding chip.

Despite having different speeds and PDU sizes, data is transmitted between the two rings without any interference. The attached devices operated smoothly without any interruptions in data transmission. Most importantly, no changes were made within the attached devices.

Chapter 10

Summary

In this thesis, a hard real-time capable ring-based wired LAN, called CarRing IV, is presented. The thesis broadly consisted of a theoretical part i.e. the new hard real-time communication protocol, and an applied part i.e. the protocol's implementation in an FPGA using a new approach. This chapter provides conclusions drawn from both parts of the thesis in its first section. Thereafter, a brief description of future work and publications is provided.

10.1 Conclusions

As described in chapter 5, the protocol provides not only hard real-time, but also isochronous real-time through its anisochronous and isochronous operation modes, respectively. Due to the protocol's access control method, those real-time guarantees do not require network-wide clock synchronization. Moreover, the protocol does not require nodes to communicate with a central node to maintain the aforementioned timing guarantees or transmit user data. Such a central node is typically referred to as a controller node and the corresponding approach is part of many actively used industrial networks.

Within the context of communication paradigms, CarRing IV supports both event- and time-triggered. Despite not requiring clock synchronization, such a feature is nevertheless implemented so as to further extend the protocol's preexisting support

for time-triggered communication, i.e. `SendAt` feature. This synchronization is based on a new decentralized approach, where the main reference node is only responsible for synchronizing the nodes within its own ring as opposed to the entire network. This eliminates the need for data-exchange pertaining to network topology status updates for the purpose of clock synchronization.

Furthermore, CarRing IV uses a reduced OSI model, which is both typical of and preferable for its application areas, e.g. automation networks. This minimalist approach also extends to its frame/packet format. Even though the protocol uses fixed-size frames/packets, the internal headers have variable structures, which are determined by the frame's type. In other words, the overall size of all headers is fixed, but the internal structure of the headers' portion of the frame/packet is variable. Since not all header fields are required by every frame/packet type, this approach maintains an overall fixed-size frames/packets as well as lowers overhead with respect to unused header bits per frame/packet type.

Another advantage of the protocol design is in the way frames are handled and transmitted. Maintaining a fixed number of continuously circulating frames within a ring is not only crucial for providing real-time guarantees, but also limits the impact of misbehaving user applications. Since each node owns a fixed number of frames within its local ring, denial-of-service attacks and noisy-neighbor problems are eliminated. In other words, no new frames are introduced within each ring after network initialization. Rather, each node transmits user data using its own frames, or another node's frames if the anisochronous mode is engaged. This also applies for routed packets, wherein only the payload and certain header fields are copied from the frame within the source ring to another preexisting frame within the destination ring.

CarRing IV's use-cases are further expanded by its transparent mode feature. In short, it transparently transfers data between user devices, which are connected to the CarRing prototype's interfaces, e.g. Ethernet, CAN. No modification to the device or user interference is required, rather the device can be directly connected and proceed to transmit data via the CarRing network using its existing interfaces. Multiple devices can be connected per prototype. To a limited extent, this feature

is comparable to the concept within the PROFINET specification that allows for the integration of existing field buses using proxy devices.

Combined with the protocol's reliable multicast, the transparent mode feature extends CarRing IV's capabilities to include field bus emulation, where CAN is regarded as the primary use-case. Furthermore, this feature allows CarRing IV to act as a backbone for other networks, which can also ease the transition of existing systems into a CarRing-based solution.

Although using a physical ring topology has the main disadvantage of being susceptible to single link failures, such a problem is both well-defined and has multiple established solutions, e.g. physical bypass for failed nodes, and hardware redundancy through dual ring topology. As explained in chapter 3, existing and actively used industrial networks use the approach of a central node to manage the network and provide timing guarantees. As such, using other topologies, e.g. mesh, is not a sufficient condition to counter single point of failures. On the contrary, the solution presented in this thesis benefits from the many advantages of using a ring topology.

The physical ring's individual node connections combined with static routing greatly supports critical traffic, which is crucial for providing timing guarantees. Furthermore, the use of a physical ring topology reducing the amount of wiring required, thereby reducing cabling complexity as well as installation and maintenance costs. Those are critical points in automotive, i.e. for CarRing IV's application areas. Whether be it through secondary ports or rings, the aforementioned hardware redundancy allows CarRing IV to be also suitable for safety critical systems.

In addition to the protocol and its design, another equally important aspect of CarRing IV is its FPGA-based hardware implementation and the approach used to realize it. Implementing all layers in hardware allows CarRing IV to be as close as possible to wire-speed, i.e. greatly improves performance while lowering overhead. The latter point refers to the overhead that arises from device drivers and intermediary interfaces used by the wide-spread and arguably standard software-hardware implementation approach. In short, lower layers (1-2) are typically implemented in hardware, while software implementation is used for higher layers (3-7).

As shown in chapter 8, the used implementation approach does not include hard- or soft-processors, i.e. no CPUs. Thus, it avoids the overhead associated with using instruction sets or OSs/kernels. In short, the approach structurally divides the protocol's core logic and supplementary operations over dedicated design components.

Each (sub-)layer's core logic is encapsulated in a stateless PE, while frames/packets are represented by arbiter/memory allocation pairs. Since those PEs are loosely coupled, this effectively removes the sequential processing that results from the direct implementation of a layered architecture. In other words, different PEs can process different frames/packets simultaneously.

Moreover, the approach is both vertically and horizontally scalable. Since the used PEs are stateless, more can be added per (sub-)layer, or the entire stack can be duplicated. The aforementioned arbiter not only manages access to the corresponding memory allocation, but also manages the execution order between PEs. Additionally, increased capacity can be easily achieved by increasing the number of arbiter/memory allocation pairs, which can be applied separately from PEs.

By having granular control and compartmentalizing every aspect of a solution, the approach lends itself to being used for implementing other software-level solutions in hardware. A solution's core logic, execution order, memory and time-sensitive operations are maintained and contained within separate dedicated design components. Within this context, CarRing IV can also be used as an example implementation.

10.2 Future Work

Despite already achieving the intended goals for the project, there is always room for further improvements. Along the work presented in this thesis, below is a list of potential directions for future work. The list is structured as follows. Each item starts with the context of the suggested update followed by a brief description.

- **Implementation approach**. In its current iteration, the approach uses arbiter/memory allocation pairs to handle processed data. Those memory allocations are static. As such, the approach can be generalized into flexible memory

allocations, wherein arbiters are further extended to handle a memory address range rather than a fixed allocation. This update accommodates systems with limited memory capacity as well as has the potential of reducing the overall memory requirements of both CarRing IV and other projects using this approach.

- **Protocol implementation**. Move the `SendAt` feature from the data-link layer to the application layer. This update reduces the complexity of the data-link layer. Consequently, it reduces the overall FPGA resources required for the design and by-extension the overall cost.

- **Protocol implementation**. Rather than having feedback interfaces per (sub-)layer, feedback is moved-to and handled-by arbiters. This update can reduce the overhead associated with feedback management, i.e. inter-layer data-exchange and matching the feedback against processed frames/packets.

- **Frames per node**. In practice, each network node's user application could have drastically level of activity and requirements. In other words, some nodes might not require or use all of their frames during network operation. Currently, this suboptimal usage of network resources is accounted-for by engaging the anisochronous mode. However, this scenario persists in the isochronous mode. As such, frames-per-node can be varied based on user application requirements, while maintaining a fixed overall number of frames per ring. Consequently, the timing behavior of the network remains unaffected. For example, one or two frames can be permanently re-allocated from node(s) with low user application requirements to other more active nodes. This update can be considered as an extension to both operation modes, i.e. a more flexible version of the modes.

- **Frame recovery**. Currently, a dropped frame can only be recreated and reintroduced into the ring by its respective owner node. By learning the pattern of incoming frames, each node is able to detect and recover dropped frames. This update is possible due to the limited number of circulating frames as well

as rings having a max size of 16 nodes. In case of a frame being dropped due to transmission errors, the corresponding node can simultaneously identify and recreate the dropped frame. This can also account for unexpected incoming frames.

- **Frame header fields**. The retransmission of acknowledged frame types is only permitted within the sender's own ring, i.e. not across rings. As such, retransmission can be tracked locally within each node, thereby eliminating the need for an explicit `Retry` field within the frame's header. Note that this does not apply to reliable multicast. Similarly, the `SendAt` header field can also be removed, since those transmissions are tracked locally within each node.

- **i-to-j feature**. Currently, a returning acknowledge can be embedded into a smaller header field, thereby allowing frame re-use via the i-to-j feature. This potential for frame re-use does not include the sender node. As such, this update allows the sender node to transmit new user data alongside the returning acknowledge. This update also permits an exception to the i-to-j transmission eligibility limitation, i.e. only `EthernetService` and routed packet types. In other words, other acknowledged frame types are also eligible for transmission via the i-to-j feature.

In addition to the above list, there are future work that are regarded more as publications.

- **CarRing IV's TSN compliance and integration**.

- **Security aspect of the CarRing IV's protocol**. As a by-product of the protocol's design, there are already security measures included, e.g. protection against denial-of-service attacks and isolated memory space, i.e. no direct access into the node's memory components is possible. The protocol is expanded further to support hardware-level encryption.

- **CarRing IV over Ethernet**. In its current iteration, the protocol's physical layer is represented by the Xilinx Aurora IP Core. In order to further broaden

and increase CarRing IV's compatibility with existing solutions, another physical layer option is added, i.e. using Ethernet PHY.

- **CarRing IV's network-wide clock synchronization**.
- **CarRing IV's reliable multicast**.
- **Updated L2 and L3 performance measurements based on the K7-prototype**. In this work, two node variations are created, one aimed at local transmissions and another at non-local transmissions. The former uses a node design that includes a reduced version of the protocol, i.e. layers 1,2 and 7. As for the latter, all protocol layers are included within the node design.

References

[1] IEC/IEEE Behavioural Languages - Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001). *IEC 61691-4 First edition 2004-10; IEEE 1364*, pages 0_1–860, 2004.

[2] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[3] ISO 11898-2:2016. Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit. International Organization for Standardization, Geneva, Switzerland, Dec 2016.

[4] ISO 11898:2015. Road vehicles - Controller area network (CAN). International Organization for Standardization, Geneva, Switzerland, Dec 2015.

[5] IEEE 1394. IEEE Standard for a High-Performance Serial Bus. IEEE Standards Association, New Jersey, United States, Jun 2008.

[6] ISO 17458:2013. Road vehicles - FlexRay communications system. International Organization for Standardization, Geneva, Switzerland, Feb 2013.

[7] ISO 17987:2016. Road vehicles - Local Interconnect Network (LIN). International Organization for Standardization, Geneva, Switzerland, Dec 2016.

[8] ISO 21806. Road vehicles - Media Oriented Systems Transport (MOST). International Organization for Standardization, Geneva, Switzerland, 2020.

[9] IEC 61784-5-14:2013. Industrial communication networks - Profiles - Part 5-14: Installation of fieldbuses - Installation profiles for CPF 14. International Electrotechnical Commission, Geneva, Switzerland, Sep 2013.

[10] 802.3cg 2019. IEEE Standard for Ethernet - Amendment 5: Physical Layer Specifications and Management Parameters for 10 Mb/s Operation and Associated Power Delivery over a Single Balanced Pair of Conductors. IEEE Standards Association, New Jersey, United States, Nov 2019.

[11] AS6003. TTP Communication Protocol. SAE International, United States, Feb 2011.

[12] AS6802. Time-Triggered Ethernet. SAE International, United States, 2016.

[13] Bernat, G. and Burns, A. and Liamosi, A. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, 2001.

[14] Black, David C. and Donovan, Jack and Bunton, Bill and Keist, Anna. *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[15] Cadence Design Systems, Inc. *Cadence C-to-Silicon Compiler, User Guide, Product Version 14.20 s400*, Jun 2016.

[16] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[17] Ethernet POWERLINK Standardization Group. *Ethernet POWERLINK Communication Profile Specification*, Nov 2018. version 1.4.0.

[18] Thomas Fuehrer, Bernd Mueller, Florian Hartwich, and Robert Hugel. Time Triggered CAN (TTCAN). In *SAE Technical Paper*. SAE International, 03 2001.

[19] EtherCAT Technology Group. EtherCAT - the Ethernet Fieldbus. Available at `https://www.ethercat.org/en/technology.html` (2020.06.20).

[20] H. Richter. *Project Description and Specification of CarRing II V16*. Feb 2013.

[21] ISO/IEC 7498. Information technology - Open Systems Interconnection - Basic Reference Model. International Organization for Standardization, Geneva, Switzerland, 1994.

[22] Brian Neil Levine and J. J. Garcia-Luna-Aceves. A Comparison of Known Classes of Reliable Multicast Protocols. In *Proceedings of the 1996 International Conference on Network Protocols (ICNP '96)*, ICNP '96, pages 112–, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP books. Taylor & Francis, 2003.

[24] Kui Liu, Sixiao Wei, Zhijiang Chen, Bin Jia, Genshe Chen, Haibin Ling, Carolyn Sheaff, and Erik Blasch. A Real-Time High Performance Computation Architecture for Multiple Moving Target Tracking Based on Wide-Area Motion Imagery via Cloud and Graphic Processing Units. *Sensors (Basel)*, 17(2):356, Feb 2017. sensors-17-00356[PII].

[25] Mall, Rajib. *Real-Time Systems: Theory and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2009.

[26] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newnes, USA, 1st edition, 2004.

[27] D. Medhi and K. Ramasamy. *Network Routing: Algorithms, Protocols, and Architectures*. The Morgan Kaufmann Series in Networking. Elsevier Science, 2007.

[28] Micrium Inc., Weston, FL 33326, USA. *uC/OS-III The Real-Time Kernal User's Manual*, 2016.

[29] MIL-STD-1553C. Interface Standard - Digital Time Division Command/Response Multiplex Data Bus. Department of Defense, United States, Aug 2020.

[30] MilCAN Working Group. *MilCAN A Specification*, May 2009. revision 3.

[31] ODVA, Inc. *The Common Industrial Protocol (CIP) and the Family of CIP Networks*, Feb 2016.

[32] ARINC 664 P7. ARINC Specification 664 - Part 7 - Avionics Full Duplex Switched Ethernet Network. ARINC Industry Activities, Maryland, United States, Sep 2009.

[33] PROFIBUS & PROFINET International. *PROFINET System Description*, Nov 2018. Order No. 4.132.

[34] Real Time Engineers Ltd. *The FreeRTOS Reference Manual*, 2016.

[35] Stock Flight Systems, Germany. *Interface specification for airborne CAN applications*, Jan 2006. revision 1.7.

[36] SystemC Language Working Group. *Functional Specification for SystemC 2.0*, 2002.

[37] SERCOS technical working groups. Technical Specifications. Available at `https://wiki.sercos-service.org/latest/Main_Page` (2020.06.20).

[38] Inc. Xilinx. What is an FPGA? Available at `https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html` (2020.10.19).

[39] Xilinx, Inc. *LocalLink Interface Specification*, SP006 edition.

[40] Xilinx, Inc. *Virtex-5 FPGA RocketIO GTX Transceiver*, UG198 (v3.0) edition, Oct 2009.

[41] Xilinx, Inc. *Aurora 8B/10B Protocol Specification*, SP002 (v2.3) edition, Oct 2014.

[42] Xilinx, Inc. *7 Series FPGAs GTX/GTH Transceivers*, UG476 (v1.12.1) edition, Aug 2018.

[43] Xilinx, Inc. *Aurora 8B/10B v11.1 LogiCORE IP Product Guide*, PG046 edition, Apr 2018.

[44] J. Zhang, L. Chen, T. Wang, and X. Wang. Analysis of TSN for Industrial Automation based on Network Calculus. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 240–247, 2019.