Chapter 2

BASIC CONCEPTS AND RELATED WORK

This chapter presents the basic concepts and terminology used in this book and gives an overview of system architectures for ultra-dependable, distributed real-time systems. Consequently, the first part of this chapter starts with an overview of distributed real-time systems and fundamental concepts of dependability in distributed real-time systems. We also present synchrony models that have been the focus of intensive theoretical studies. The second part describes event-triggered and time-triggered systems and relates these control paradigms to prevalent computational models. This chapter ends with a discussion of distributed system architectures for ultra-dependable systems and relates each architecture to the event-triggered and time-triggered computational models.

2.1 Distributed Real-Time Systems

A real-time computer system is a computer system in which correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instants at which these results are produced [Kopetz, 1997, p. 2]. The real-time computer system and the enclosing environment form a larger system called the *real-time system.* The real-time computer system must react to stimuli from the controlled object in the environment within time intervals specified via *deadlines.* If a result has utility even after the deadline has passed, the deadline is classified as *soft.* Deadlines are called *hard deadlines,* if catastrophic consequences can result from missing a deadline. A real-time computer system that must meet at least one hard deadline is called a *hard real-time computer system* or a *safety-critical real-time computer system* [Kopetz, 1997]. In a soft real-time computer system no catastrophic consequences arise through deadline violations.

If the real-time computer system is distributed, it consists of a set of nodes interconnected by a real-time communication system. Each node can be partitioned

into at least two subsystems, a communication controller and a host computer. The interface between the communication controller and the host computer is called the *communication network interface.* The set of communication controllers and the common communication network form the *communication system.* The purpose of the communication system is the transport of messages from the CNI of a sender to the CNIs of the receivers.

Component and Component Interfaces

In many engineering disciplines, the term component refers to a building block in the design of a larger system. In the context of distributed embedded real-time systems, a complete node seems to be the best choice for a component [Kopetz, 1998a], since the component-behavior can then be specified in the domains of value and time. Thus, a component is considered to be a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system). The integration of components into a larger system is realized via a communication service that enables components to exchange messages across their linking interfaces (LIFs) [Kopetz and Suri, 2003]. The linking interface reduces a component functional and temporal description of those services that are required for the integration. The internal structure of the component is encapsulated and hidden from the user.

Linking interfaces are specified through an operational and a meta-level specification. The operational specification consists of the syntactic specification and the temporal specification. The syntactic specification defines the structure and the name of the exchanged messages. The temporal specification defines the temporal sequence of message exchanges. The meta-level specification defines the semantic by assigning meaning to information.

The LIF can be decomposed into a service requesting linking interface (SRLIF) and a service providing linking interface (SPLIF). A component offers its services to its environment via the SPLIF. The SRLIF enables a component to exploit services provided by other components. A component can depend on these services for being able to offer its own services via the SPLIF.

Concepts of Time, State, and Event

The concepts of time and state are fundamental in computer science. A common concept of time is a prerequisite for the consistent specification of time-out values, which are required in many communication protocols. In real-time systems, the validity of real-time information depends on the progression of physical time. For example, it makes little sense to use the measured angular position of a crankshaft in an automotive engine, if the precise instant when this position was measured is not recorded as part of the measurement [Jones et al., 2002]. In dependable systems,

concepts of time and state are necessary in the construction of mechanisms for the detection and handling of failures. The detection of even the simplest external failure mode of a component, a crash failure [Laprie, 1992], depends on error detection in the temporal domain. The masking of faults by voting requires a consistent notion of state in replicas.

Concept of Time

In the context of real-time systems a time model based on Newtonian absolute time seems to be the best choice. In this model, the continuum of real-time can be modeled by a directed timeline consisting of an infinite set of instants [Whitrow, 1990]. In a distributed computer system, nodes capture the progression of time with physical clocks containing a counter and an oscillation mechanism. A physical clock partitions the time line into a sequence of nearly equally spaced intervals, called the micro granules of the clock, which are bounded by special periodic events, the ticks of the clock. An observer can record the current granule of the clock to establish the *timestamp* of an occurrence.

Since any two physical clocks will employ slightly different oscillators, the time-references generated by two clocks will drift apart. Clock synchronization is concerned with bringing the time of clocks in a distributed system into close relation with respect to each other. A measure for the quality of clock synchronization is the precision, which is defined as the maximum offset between any two clocks during an interval of interest. By performing clock synchronization in an ensemble of local clocks, each node can construct a local implementation of a global notion of time. The availability of such a synchronized global time [Lamport, 1984] simplifies the solution of agreement problems, because nodes can establish a consistent temporal order of events based on timestamps.

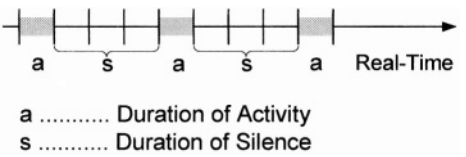

Figure 2.1. Sparse Time Base

However, due to the synchronization and digitalization error it is impossible to establish the temporal order of occurrences based on their timestamp, if timestamps differ by only a single tick. A solution to this problem is the introduction of a *sparse time base* [Kopetz, 1992], where time is partitioned into an infinite sequence of alternating durations of activity and silence. The activity intervals establish a system-wide action lattice (see Figure 2.1). All occurrences within a specified duration of activity of the action lattice are considered to happen at the same time. If the lattice points of this action lattice are properly chosen, temporal order and

simultaneity of occurrences can be realized based on timestamps without having to execute an agreement protocol.

While it is possible to restrict occurrences within the sphere of control of the computer system to these activity intervals, this is not possible for occurrences in the environment. Such occurrences happen on a dense time base and must be assigned to an interval of activity by an agreement protocol in order to get a system-wide consistent perception of when an occurrence happened in the environment.

Concept of State

The notion of state is fundamental for the investigation of complex systems. *State* is introduced in order to separate the past from the future. This book applies the definition of state by Mesarovic and Takahara [Mesarovic and Takahara, 1989]. This definition is based on the idea that *if one knows what state the system is in, he could with assurance ascertain what the output will be* [Mesarovic and Takahara, 1989, p. 45]. Hence, the state of a system embodies all past history of the given system and concentrates knowledge about occurrences of the past. In a deterministic system, future outputs solely depend on the current state and the future inputs.

Concept of Event

An *event* is a change of state, occurring at an instant [Nance, 1981]. An event is said to be determined, if the condition on event occurrence can be expressed strictly as a function of time. Otherwise, the event is contingent.

Concepts of Interface State and Function

The *interface state* [Jones et al., 2002] is part of the state of a component, namely the state as viewed from a particular interface. We define a *function* as an application specific input/output transformation that operates on the interface state of components. The function of a complete computer system is specified via the transformations of the state as viewed from the instrumentation interface. The function of a component are the transformations of the state as viewed from the linking interfaces.

Real-Time Entities, Objects, and Images

Real-time entities [Kopetz and Kim, 1990] are state variables of relevance. Real-time entities model the dynamics of a real-time application and change their state as time progresses. A real-time entity is located either in the environment or the controlling computer system. Every real-time entity is in the sphere of control of a subsystem that has the authority to set the value of the real-time entity. Outside its sphere of control, a real-time entity can only be observed.

A container within the computer system for holding a real-time entity is called *real-time object* [Kopetz and Kim, 1990]. In case of a distributed real-time object,

every local site has its own replicated version of the real-time object. An example for such a distributed real-time object is global time, which provides the consistency constraint that at any point of real-time the clock value read by two nodes differs by at most one clock tick.

A *real-time image* is stored in a real-time object and provides the current picture of a real-time entity. The validity of a real-time image is time-dependent, which can be represented by the concept of *temporal accuracy* [Kopetz, 1997, p. 103]. A real-time image is temporally accurate, if its value is a member of the set of values that the real-time entity had in its recent history. The length of the recent history is denoted as *temporal accuracy interval.* The dynamics of a real-time entity determine the admissible temporal accuracy interval.

Types of Temporal Control Signals

A trigger is a control signal that initiates an action in the controlling computer system, like the execution of a task or the transmission of a message [Kopetz, 1993]. Depending on the source from which a trigger is derived, one can distinguish event triggers and time triggers.

Event Triggers

An *event trigger* is a control signal that is derived from an event, i.e. a state change in a real-time entity. The event can originate either from activities within the computer system (e.g., termination of a task) or from state changes in the natural environment (e.g., alarm condition indicated by a sensor element). In the latter case, the event trigger serves as a mechanism by which the environment delivers a service request to the controlling computer system. In general, such a service request will start a sequence of computational and communication activities.

Time Triggers

A *time trigger* is a control signal that is generated at a particular point in time of a synchronized global time base. Time triggers are solely derived from the progression of global time, which is a distributed real-time entity established by the clock synchronization service. The set of time triggers is a subset of the set of event triggers, since time triggers correspond to a particular class of events, namely changes in the state of global time. Time triggers are discriminated from other event triggers, since systems restricting control signals to time triggers offer properties that are desirable for distributed real-time systems. Among these properties are temporal predictability and composability [Kopetz, 1995b].

2.2 Concepts of Dependability

Dependability is the ability of a computing system to deliver services that can justifiably be trusted [Carter, 1982]. The service delivered by a system is its behavior

as it is perceptible by another system (human or physical) interacting with the former [Laprie, 1992].

Dependability Threats – Failure, Error, Fault

A *failure* [Laprie, 1992] occurs when the delivered service deviates from fulfilling the functional specification. An *error* is that part of the system state which is liable to lead to a subsequent failure. A failure occurs when the error reaches the service interface. A *fault* is the adjudged or hypothesized cause of an error. As stated in [Avizienis et al., 2001], the concept of fault is introduced to stop recursion.

Due to the recursive definition of systems, a failure at a particular level of decomposition can be interpreted as a fault at the next upper level of decomposition, thereby leading to a hierarchical causal chain.

Fault Containment and Error Containment

A *fault containment region (FCR)* is defined as a subsystem that operates correctly regardless of any arbitrary logical or electrical fault outside the region [Lala and Harper, 1994]. The justification for building ultra-reliable systems from replicated resources rests on an assumption of failure independence among redundant units. For this reason the independence of FCRs is of critical importance [Butler et al., 1991]. The independence of FCRs can be compromised by shared physical resources (e.g., power supply, timing source), external faults (e.g., EMI, spatial proximity) and design.

Although an FCR can restrict the immediate the impact of a fault, fault effects manifested as erroneous data can propagate across FCR boundaries. For this reason the system must also provide error containment [Lala and Harper, 1994] to avoid error propagation by the flow of erroneous messages. The error detection mechanisms must be part of different FCRs than the message sender [Kopetz, 2003]. The *error containment region (ECR),* i.e. the set of FCRs that perform error containment, must consist of at least two independent FCRs. Otherwise, the error detection mechanism may be impacted by the same fault that caused the message failure.

Fault Hypothesis

The *fault hypothesis* specifies assumptions about the types of faults, the rate at which components fail and how components may fail [Powell, 1992]. The *assumption coverage* is the probability that these assumption hold in reality. Since fault-tolerance mechanisms of a system are based on these assumptions, the complete system can fail in case the assertions concerning faults, failure rates, and failure modes are violated.

Failure modes of components are defined through the effects as perceived by the service user, i.e. independently of the actual cause or rate of failures. Formally, a failure mode is defined in terms of an assertion on the sequence of value-time tuples

that a failed or failing system is assumed to deliver [Powell, 1992]. Failure modes determine the degree of redundancy required to ensure correct error processing.

Based on the rigidity of assumptions, the following hierarchy of failures modes can be established [Cristian, 1991b]:

- **Fail-stop failures:** A fail-stop failures is defined as a component behavior, where the component does not produce any outputs. The component omits to produce output to subsequent inputs until it restarts. It is additionally assumed that all correct component detect the fail-stop failure.
- **Crash Failures:** A component suffering a crash failure does not produce any outputs. In contrast to fail-stop failures, a crash failure can remain undetected for correct components.
- **Omission Failures:** An omission failure occurs, if the sender component fails to send a message, or the receiver fails to receive a sent message. As a consequence, the receiver does not respond to an input. An omission failure can remain undetected for correct components.
- **Timing Failures:** The component does not meet its temporal specification. Outputs of a component are delivered too early or too late.
- **Byzantine or Arbitrary Failures:** There is no restriction on the effects a service user may perceive. Arbitrary failures include the forging of messages and "two-faced" component behaviors [Lamport et al., 1982].

2.3 Degrees of Synchrony

Existing models of distributed systems differ in their inherent notion of real-time. This is usually expressed in certain assumptions about process execution speeds and message delivery delays.

Asynchronous and Synchronous Systems

According to [Verissimo, 1997] the degree of synchrony of a system is defined by the system's compliance to five conditions:

S1 bounded and known processing speed

S2 bounded and known message delivery delay

S3 bounded and known local clock drift rate

S4 bounded and known load pattern

S5 bounded and known difference of local clocks

In *synchronous systems,* all five conditions are satisfied. In particular, there are known upper bounds of the durations of communication and processing activities.

The synchronous system model reduces the complexity of the design and implementation of dependable distributed applications, because non faulty processes can exploit the progression of time to predict each others' progress. For this reason, fault-tolerant systems for safety-critical control applications are usually based on the synchronous approach, although there are differences in the extent to which the basic mechanisms of the system guarantee the satisfaction of the synchrony assumption [Rushby, 1999b]. A high assumption coverage of the temporal bounds of the basic synchrony conditions is crucial, since synchronous systems are prone to incorrect behavior, if the implementation violates these timing constraints.

In the absence of such bounds, we speak of an *asynchronous system* [Fischer et al., 1985]. The asynchronous model contains the weakest assumptions, according to Schneider asynchrony is a "non-assumption" [Schneider, 1993]. Hence, an algorithm that works in the asynchronous model also works in all other models of synchrony. A fully asynchronous model has limitations restricting its usefulness in practical systems. Many problems of interest do not have deterministic solutions, as proven via impossibility results [Fischer et al., 1985; Lynch, 1989]. In particular, an asynchronous model does not allow timeliness specifications. In case of crash failures, it is impossible to guarantee to solve consensus within a bounded time. The main problem in the asynchronous model is the impossibility of distinguishing a slow processor from a failed processor.

Many existing systems are based on a model with an intermediate level of synchrony. For non safety-critical applications, probabilistic or partial satisfaction of synchrony assumption is often accepted for economic reasons. Examples of intermediate synchrony models are the timed asynchronous system model [Cristian and Fetzer, 1999], and the quasi-synchronous model [Verissimo and Almeida, 1995]. The *timed asynchronous system model* makes weak assumptions about the underlying infrastructure, more precisely it assumes no communication bounds and no global time. In a *quasi-synchronous* system a subset of the five synchrony conditions is satisfied only probabilistically, i.e. there is a known probability that an assumed bound does not hold.

Consensus Problem

Fundamental to cooperation in a distributed system is the ability of agreeing on a quantum of information in order to reach common decisions and to maintain the integrity of the system [Barborak et al., 1993, p. 171]. In the *consensus problem,* all correct processes propose a value and must reach a unanimous and irrevocable decision on some value that is related to the proposed values [Fischer, 1983]. The unanimity requirement ensures that in case all proposed values are identical, the consensus value is identical to this proposed value. Furthermore, the consensus value should depend on the initial values (non-triviality).

Two problems closely related to consensus are the *interactive consistency problem* [Pease et al., 1980] and the *byzantine generals problem* [Lamport et al., 1982]. The *interactive consistency problem* requires non-faulty processes to agree on a consistent vector called the consensus vector y. The ith element of y has to be the value proposed by process i, if i is non-faulty (validity). In the Byzantine Generals problem, a sender attempts to send a value to all other processors. All correct processes have to reach consensus on this value, if the sender is correct (strong generals problem). The weak generals problem only requires consensus, if no failure occurs during the protocol execution.

For the asynchronous system model, Fischer et. al proved that consensus is impossible, even if only one processor crashes during the protocol execution [Fischer et al., 1985]. This impossibility result is important because of the equivalence of the consensus problem with several other important problems. In [Fischer, 1983] the generals problem and interactive consistency are reduced to the consensus problem. The equivalence of the atomic broadcast problem and the consensus problem is shown in [Chandra and Toueg, 1996].

Failure Detectors

Inspired by the impossibility result of Fischer et al. [Fischer et al., 1985] research has been focused on the question how much synchrony is needed to achieve consensus in a distributed system [Dolev et al., 1987].

Chandra and Toueg [Chandra and Toueg, 1996] proposed an alternative approach to circumvent the impossibility result. In order to decide whether a process has actually crashed or is only very "slow", they augmented the asynchronous model of computation with *unreliable failure detectors,* i.e. an external failure detection mechanism that can erroneously indicate that a component has failed, only to correct the error at a later time. A distributed failure detector comprises a set of local failure detector modules. Each module maintains a list of processes which are currently suspected to have crashed and can add or remove processes from this list. Furthermore, the lists of suspects of two local failure detector modules can be different at any given time.

Failure detectors can be characterized according to the *completeness* and *accuracy* properties. Roughly speaking, completeness requires that every crashed process is eventually suspected and accuracy restricts the mistakes a failure detector can make.

In [Chandra et al., 1996] it has been shown that in order to solve *consensus,* any failure detector has to provide at least as much information as the *eventually weak failure detector* $\Diamond\mathcal{W}$. The definition of a class of failure detectors must be seen as a specification of the failure detection mechanism [Guerraoui and Schiper, 1997]. An optimal algorithm that implements the weakest failure detector is described in [Larrea et al., 2000].

2.4 Communication System Paradigms

Event-triggered and time-triggered communication systems are two different paradigms for the construction of the communication infrastructure of a distributed real-time system. The major difference between these paradigms lies in the location of control. Event-triggered communication systems are based on external control via event triggers. The decision when a message is to be transmitted is within the sphere of control of the application software in the host (see Figure 2.2). In a time-triggered communication system, the communication controller decides autonomously about the global points in time at which messages are transmitted. This autonomous control of a time-triggered communication system results from restricting temporal control signals to time triggers, which are independent of the state changes in the environment and the host computer.

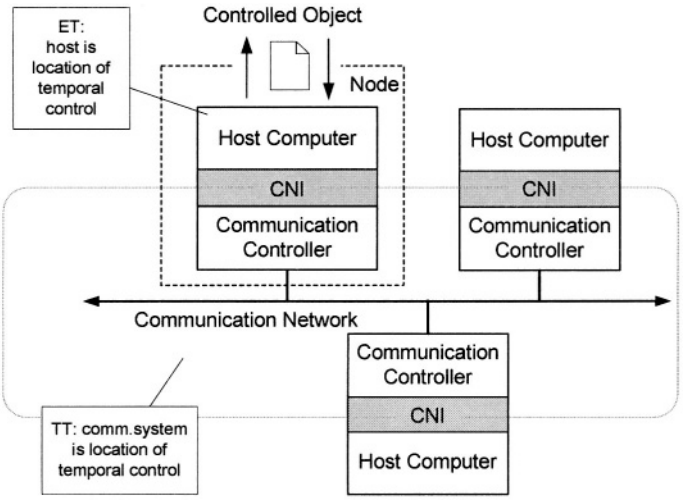

Figure 2.2. Autonomous and External Control at the Communication System

This section compares event-triggered and time-triggered communication systems and presents distinctive features of the communication network interface in these two paradigms. Furthermore, we will discuss the data and control flow via the communication system and the requirements for an underlying transport protocol. The comparison of the two paradigms also includes a comparative evaluation of event-triggered and time-triggered communication systems.

Event-Triggered Communication Systems

An event-triggered communication system is designed for the sporadic exchange of event messages, combining event semantics with external control. At the sender side, these event messages are passed to the communication system via an explicit

transmission request from the host application (external control) or as a result of the reception of a request message (e.g., client/server interaction). At the receiver side, the host application either fetches the incoming message from the communication system (polling for messages) or the communication system presses received messages into the host application (interrupt mechanism).

Supported Information Semantics

An event-triggered communication system can support the transmission of information with *event semantics* and/or *state semantics* [Kopetz, 1997]. Information with state semantics contains the absolute value of a real-time entity (e.g., temperature in the environment is 41 degrees Celsius). Since applications are often only interested in the most recent value of a real-time entity, state information allows the communication system to overwrite old state values with newer state values.

Information with event semantics relates to the occurrence of an event. Event information represents the change in value of a real-time entity associated with a particular event. Messages containing event information transport relative values (e.g., increase of the temperature in the environment by 2 degrees). In order to reconstruct the current state of a real-time entity from messages with event semantics, it is essential to process every message exactly-once. The loss of a single message with event information can affect state synchronization between a sender and a receiver.

Communication Network Interface

The Communication Network Interface (CNI) is the most important interface in a distributed real-time system [Kopetz, 1997], because it determines the abstraction a node forms in the context of a distributed real-time system.

We can classify event-triggered communication systems based on the direction of the control flow relative to the data flow at the CNI. Depending on the direction of the control flow relative to the data flow, one can distinguish an information push and an information pull behavior in the transfer of information between the host computer and the communication controller. In an *information push behavior* [DeLine, 1999], data and control flow have the same orientation, i.e. the information transfer occurs via the sender's request. An *information pull behavior* starts an information transfer via the receiver's request, data and control flow have complementary orientations.

- **Information Push for Information Flow from Host to Communication Controller:** The use of the information push principle for a host's message transmission requests is the prevalent type of control flow at the CNI in event-triggered communication systems. Thereby, the event-triggered system supports an event-based design, in which computational and communication activities a triggered by external stimuli and by the progression of computational activities.

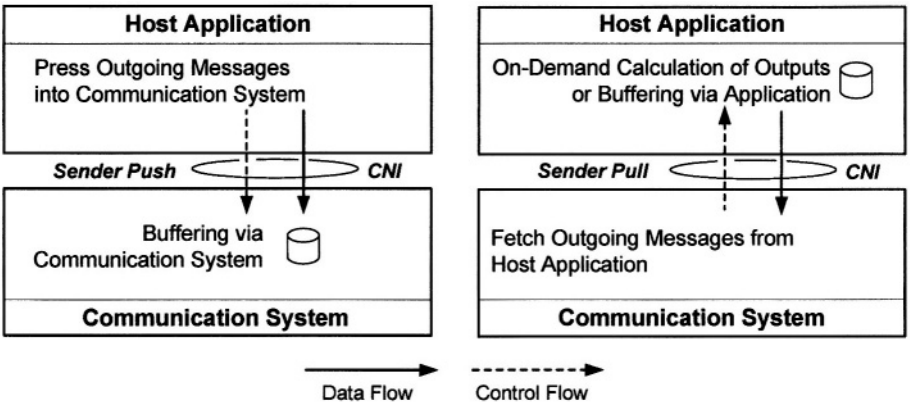

Figure 2.3. Sender Push and Sender Pull at CNI for Outgoing Messages

As depicted in Figure 2.3, the application in the host computer presses messages into the communication system, e.g., requesting a message dissemination via a "send message" primitive. The information push principle for sending activities is therefore ideal for the host application, since the host application determines the points in time when data is transferred to the communication system.

This type of control also enables a generic buffer management for outgoing messages via the communication system. The need for buffering of messages arises, in case outgoing messages aggregate, e.g., due to the fact that the common network is occupied by other nodes or an application temporarily requests messages at a rate exceeding the available network bandwidth.

Although the buffer management functionality is application independent, the dimensioning of the intermediate structures at the communication system requires knowledge about interarrival times of messages from the host application in order to prevent buffer overflows. In case of an unrestricted rate of message transmission requests from the host computer, these intermediate data structures cannot be dimensioned and a potential overload of the communication system is inevitable.

- **Information Pull for Information Flow from Host to Communication Controller:** An information pull mechanism for outgoing messages at the CNI means that the communication system fetches outgoing message from the host application. An example for such a communication behavior occurs at the server side in a client/server interaction. A server object is a passive entity, which requires an external trigger (e.g., via the communication system) that results in computational activities and subsequently causes the dissemination of a response message.

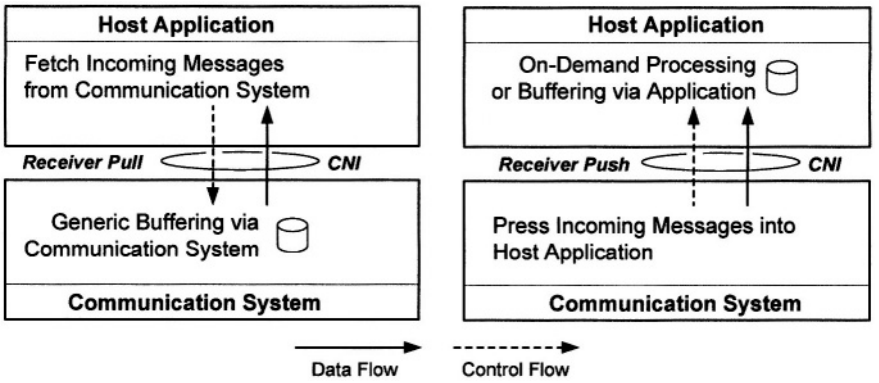

Figure 2.4. Receiver Push and Receiver Pull at CNI for Incoming Messages

Many programming languages for distributed systems provide support for such control patterns via synchronous message passing [Bal et al., 1989], i.e. a blocking send primitive that blocks until the communication partner has executed a corresponding receive statement.

- **Information Push for Information Flow from Communication Controller to Host:** An information push for message receptions means that the communication controller presses messages received via the network into the host application (see Figure 2.4). This control type for receptions represents the well-known interrupt mechanism. An information push CNI does not require restrictions of message interarrival times at the level of the communication system, since the communication system immediately forwards received messages to the host computer. The handling of the incoming message load is therefore shifted to the application software in the host computer. The main problem of this approach is the fact that the temporal behavior of the host application depends on the behavior of other applications that send messages.

- **Information Pull for Information Flow from Communication Controller to Host:** An information pull interface at the CNI requires the application in the host computer to fetch messages from the communication system. This is ideal for the application in the host computer, since it can determine the points in time when messages are retrieved and processed via the CNI.

  The information pull mechanism prevents an interruption of application software through incoming messages. Incoming messages are buffered in intermediate datastructures and explicitly fetched by the application software. Similar to the information pull behavior for message transmissions, generic message buffering capabilities of the communication system are possible. Hence, incoming messages do not automatically lead to the consumption of computational resources.

The application can defer the processing of messages in case of high load. Nevertheless, the message buffering requires knowledge about the interarrival times of messages from the network and the service times of the application software. If for intervals of limited duration, the interarrival times of messages are smaller than the service times of messages at the host computer, the communication system can buffer messages in intermediate data structures. However, in case of unrestricted interarrival or service times, these intermediate data structures cannot be dimensioned and messages can be lost. The resulting message omission failures can affect state synchronization, if messages contain information with event semantics.

Transport Protocol

In an event-triggered communication system, the transport protocol must support the transmission of event messages with exactly-once delivery semantics in order to be able to maintain state synchronization. For preserving exactly-once delivery semantics in the presence of transient failures (e.g., omission or corruption of messages), many event-triggered transport protocols employ acknowledgments and timeouts in combination with unique identifiers or sequence numbers assigned to messages.

In a positive acknowledgment scheme, the receiver explicitly informs the sender about the successful reception of the message, which is identified in the corresponding acknowledgment message (e.g., via the sequence number). If the sender does not receive an acknowledgment message within a reasonable timeout, it will retransmit a message k times. Protocols using such an acknowledgment scheme are denoted as Positive-Acknowledgment-or-Retransmission (PAR) protocols. A major difficulty in the design of a PAR protocol is the calculation of the timeout, when delays of underlying networks are variable and dynamic [Iren et al., 1999]. In many protocols, the determination of timeouts is based on round-trip time estimations [Karn and Partridge, 1988].

A negative acknowledgment mechanism explicitly identifies messages that have not been received. Since only the sender has knowledge about the points in time when a message has to be transmitted, a negative acknowledgment is only possible when a receivers detects missing sequence numbers when subsequent messages are received. When a sender is not required to send messages in regular intervals, a negative acknowledgment scheme leads to unbounded error recovery times.

In broadcast communication relationships, the construction of distributed processing algorithms can be significantly simplified, if the event-triggered transport protocol provides a globally consistent delivery ordering. The corresponding atomic broadcast protocols operate between the application programs and the broadcast network and isolate the application programs from the unreliable characteristics of the communication network. However, most atomic broadcast protocols involve multiple rounds of message exchanges and introduce significant temporal

uncertainty [Kopetz and Kim, 1990]. For example, the reliable broadcast protocols described in [Chang and Maxemchuk, 1984] add latency while forwarding the message to a node that has the permission to establish broadcast orderings. In [Cristian et al., 1985] the delivery latencies depend on the transmission latencies between nodes and the precision of clock synchronization.

Flow Control

Flow control is concerned with balancing the message production rate of the sender with the message consumption rate of the receiver, such that the receiver can follow the sender. A receiver will be unable to process incoming messages in case of unconstrained message arrival rates. The aggregation of event messages in intermediate data structures (e.g., incoming messages queues) will eventually cause omission failures. The purpose of flow control is the prevention of such an overload condition.

Explicit flow control is the predominant type of flow control in event-triggered communication systems. The receiver exerts back pressure on the sender by sending explicit acknowledgment messages. An acknowledgment message informs the sender about the receiver's ability to receive further information.

In a communication system employing explicit flow control, a unidirectional data flow involves a bidirectional control flow. Such an interface is called a *composite interface* [Kopetz, 1999a]. The transmission of a message through the sender depends on a control flow in the opposite direction, i.e. from the receiver to the sender. While a composite interface prevents senders from overloading receivers, correctness of a sender depends on correctness of receivers, which can constitute a problem with respect to error propagation. Furthermore, explicit flow control cannot be applied for events occurring in the natural environment, because it is usually impossible to exert back pressure on the natural environment. Explicit flow control is based on the assumption that sender is within sphere of control of the receiver.

Time-Triggered Communication Systems

A time-triggered communication system is designed for the periodic transmission of state information. It initiates all communication activities at predetermined global points in time. Hence, the temporal behavior of the communication system is controlled solely by the progression of time.

Supported Information Semantics

A time-triggered communication system is designed for the periodic exchange of messages carrying state information. These messages are called *state messages.* The self-contained nature and idempotence of state messages eases the establishment of state synchronization, which does not depend on exactly-once processing guarantees. Since applications are often only interested in the most recent value

of a real-time object, old state values can be overwritten with newer state values. Hence, a time-triggered communication system does not require message queues.

Communication Network Interface

As depicted in Figure 2.5 the CNI of a time-triggered communication system acts as a temporal firewall [Kopetz and Nossal, 1997]. The sender can deposit information into the CNI according to the information push paradigm, while the receiver must pull information out of the CNI. A time-triggered transport protocol autonomously carries the state information from the CNI of the sender to the CNIs of the receivers. Since no control signals cross the CNI, temporal fault propagation is prevented by design.

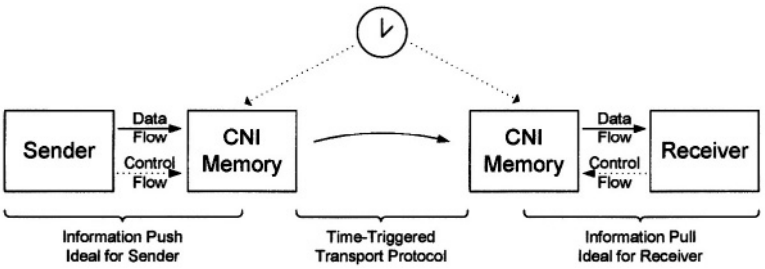

Figure 2.5. Data Flow (Full Line) and Control Flow (Dashed Line) of a Temporal Firewall Interface [Kopetz, 2001]

The state messages in the CNI memory form two groups. Those state messages that are written by the host application represent the node's service providing linking interface (SPLIF). The communciation controller reads these messages and disseminates them during the slots reserved for the node via the underlying TDMA scheme. Those messages that form the node's service requesting linking interface (SRLIF) are written by the communication controller and read by the host application.

Consistency of information exchanged via the CNI can be ensured by exploiting the a priori knowledge about the points in time when the communication system reads and writes data into the CNI. The host application performs implicit synchronization by establishing a phase alignment between its own CNI accesses and the CNI accesses of the communication controller. A different approach is the use of a synchronization protocol, such as the *Non-Blocking Write Protocol* [Kopetz and Reisinger, 1993].

Transport Protocol

The media access control strategy of a time-triggered communication system is Time Division Multiple Access (TDMA). TDMA statically divides the channel capacity into a number of slots and assigns a unique slot to every node. The communication activities of every node are controlled by a time-triggered communication

schedule. The schedule specifies the temporal pattern of messages transmissions, i.e. at what points in time nodes send and receive messages. A sequence of sending slots, which allows every node in an ensemble of n nodes to send exactly once, is called a TDMA round. The sequence of the different TDMA rounds forms the cluster cycle and determines the periodicity of the time-triggered communication.

The a priori knowledge about the times of message exchanges enables the communication system to operate autonomously. The temporal control of communication activities is within the sphere of control of the communication system. Hence, the correct temporal behavior of the communication system is independent of the temporal behavior of the application software in the host computer and can be established in isolation.

Flow Control

Time-triggered communication systems employ *implicit flow control* [Kopetz, 1997]. Sender and receiver agree a priori on the global points in times when messages are exchanged. Based on this knowledge, a component's ability for handling received messages can be ensured at design time, i.e. without acknowledgment messages. Implicit flow control is well-suited for multicast communication relationships, because a unidirectional data flow involves only a unidirectional control flow (*elementary interface* [Kopetz, 1999a]).

Comparative Evaluation

The strengths of event-triggered communication systems are flexibility and a high resource utilization, which enables cost-effective solutions for best-effort systems (e.g., body electronics in a car). Time-triggered communication systems, on the other hand, can provide temporal predictability, composability, and replica determinism. In addition, the a priori knowledge about the points in time of communication activities facilitates the construction of error detection and error containment mechanisms. Consequently, the time-triggered communication paradigm is being accepted for the communication infrastructure of safety-critical applications [Rushby, 2001b]. Examples of time-triggered communication protocols for safety-critical applications are SAFEbus [Hoyme and Driscoll, 1993] (avionic domain), FlexRay [R. Mores et al., 2001] (automotive domain), and TTP [Kopetz and Grünsteidl, 1994] (avionic and automotive domains).

Temporal Predictability

In an event-triggered communication system, the transmission requests of host applications determine the temporal patterns of message exchanges. If multiple host applications request message transmissions in rapid succession (e.g., during peak load scenarios), contention on the network will require the communication system to delay messages. This relationship between communication system load

and message transmission latencies can result in significant communication jitter. Another source of communication jitter are retransmission mechanisms employed for handling transient communication failures. Retransmission mechanisms also tend to result in a trashing behavior, i.e. abruptly decreasing throughput with increasing load. If the communication system delays messages, because it can barely handle the message load, a retransmission mechanism will generate additional load when local timeouts elapse.

A time-triggered communication system is based on a static temporal control structure. Since transmission latencies are independent of the event occurrences in the environment and the controlling computer system, the latency jitter is minimized.

Composability

Since an event-triggered communication system is based on external control, the temporal behavior of an event-triggered communication system depends on the points in time of the transmission requests from host applications. Hence, there is an inseparable interrelationship between the compliance to the temporal specification of linking interfaces between components and the component implementations. Compared to time-triggered systems, event-triggered systems provide a weaker separation of architecture and component design.

Time-triggered communication systems support an exact specification of linking interfaces, both in the value domain and time domain, which is a prerequisite for temporal composability [Kopetz and Obermaisser, 2002]. The autonomous control of the communication system isolates the temporal behavior of the communication system from the behavior of the host computers.

Replica Determinism

Fault-free replicated components exhibit replica determinism [Poledna, 1995], if they deliver identical outputs in an identical order within a specified time interval. Replica determinism simplifies the implementation of fault-tolerance by active redundancy, since failures of components can be detected by carrying out a bit-by-bit comparison of the results of replicas. Replica nondeterminism is introduced either by the interface to the real world or the system's internal behavior.

In an event-triggered communication system, a major source for replica nondeterminism are the timeouts of retransmission and acknowledgment mechanisms. If local timeouts are used without global coordination, some replicas can decide locally to timeout, while others will not due to slightly different processing speeds. A further source of replica nondeterminism are inconsistent inputs. If the communication systems does not guarantee a consistent and ordered delivery of input messages, replicas are likely to produce inconsistent output values.

Flexibility

The major strengths of event-triggered communication systems lie in the higher degree of flexibility and the support for a dynamic allocation of resources, which is attractive for variable resource demands. Compared to time-triggered communication systems, an event-triggered communication systems provides fewer a priori restrictions concerning the temporal behavior of nodes. The low static dependencies among components are important, if it is undesirable to make restrictions at design time regarding the addition, removal and replacement of components. An event-triggered communication system eases the migration of functionality between nodes and the addition of functionality that was not anticipated during the system design (extensibility). However, these activities can require the retesting of the temporal behavior of the system, since an event-triggered communication system with external control does not encapsulate the effects of changes in a node.

In a time-triggered communication system, communication slots must either be reserved beforehand in order to allow future extension, or a new communication schedule must be constructed when nodes are added.

Error Detection

In an architecture without replication, error detection is only possible by comparing the actual behavior of a node to some a priori knowledge about the expected behavior. In an event-triggered communication system, the limited a priori knowledge about the communication activities complicates error detection. For example, if a node is not required to send a message at regular intervals, it is not possible to detect a node failure within a bounded latency [Kopetz, 1997].

In a time-triggered communication system the periodic message send times are membership points of the sender [Kopetz et al., 1991]. Every receiver knows a priori when a message of a sender is supposed to arrive, and interprets the arrival of the message as a life sign at the membership point of the sender. From the arrival of the expected messages at two consecutive membership points, it can be concluded that the sender was operational during the interval delimited by these membership points.

Resource Utilization

In an event-triggered communication system, network bandwidth is required only for those messages, for which dissemination is requested by the host applications. Bandwidth that is not consumed by a node is available to other nodes, thus enabling the statistical multiplexing of bandwidth. This approach allows the dimensioning of resources according to average demands, which is desirable for non critical communication activities, where a resource-adequacy policy would not be cost-effective.

In a time-triggered communication system all communication activities are fixed and planned for the specified peak load demand. Hence, the resource utilization of an event-triggered communication system will be better than that of a comparable time-triggered communication system, if load conditions are low or average. However, for safety-critical applications a resource adequacy policy is required anyway in order to guarantee safety during peak load scenarios.

Specification of Interfaces during the Design Process

The design of distributed embedded systems is often viewed as a top-down approach, which proceeds via well-defined levels of abstraction [Pires et al., 1993]. The specification serves as the input description for the designer. The specification provides the definition of the common behavior of the system by describing the interactions between the system and its environment in the temporal and value domain. The specification is transformed into a functional description, which is given as a set of explicit or implicit relations which involve inputs, outputs and possibly internal information [Edwards et al., 1997]. The functional specification is mapped to an architecture by performing a decomposition of functions and assigning resulting functional units to components.

This section compares differences in the design process of distributed real-time systems, depending on whether a functional specification is mapped to an event-triggered or a time-triggered system architecture. The major difference between event-triggered and time-triggered architectures lies in the level of rigidity in the temporal specification of linking interfaces during the mapping of a functional specification to an architecture. A time-triggered system specifies the precise points in time of information exchanges with respect to a global time base. Event-triggered systems can establish flexibility and high average performance by employing a weaker temporal specification of interfaces. Examples for such temporal specifications are deterministic or probabilistic message interarrival and service times.

Time-Triggered Systems

The main design principle of a time-triggered system is the separation of local and global concerns. Global concerns are the subject of architecture design, during which the decomposition of the system into components and the specification of the component interfaces is performed. Component design focuses on local concerns of the application in a node, such as task scheduling.

If a system can be developed without legacy components, then a top-down decomposition will be pursued. During the architecture design phase the application is decomposed into a set of clusters and nodes. A systematic fault-tolerance concept is devised for internal physical faults of nodes. Sets of nodes that compute the same function are logically grouped into fault-tolerant units. A fault-tolerant unit will tolerate the failure of any of its independent constituent nodes without a degradation of service. The number of nodes required for the construction of a fault-tolerant

unit depends on the assumed failure modes. For example, the consistent failure of a single node can be masked by a fault-tolerant unit consisting of three nodes (triple modular redundancy). The nodes operate in replica determinism [Poledna, 1995] and present their results to a voter (located at every consumer of the result) that makes a majority decision.

After the decomposition has been completed, the CNIs of nodes are specified in the value and time domain. The state variables that are to be exchanged via the time-triggered communication system are identified and the admissible temporal accuracy intervals [Kopetz and Kim, 1990] are determined. The size of a temporal accuracy interval is determined by the dynamics of the real-time entities in the controlled object. Given these data, a cluster compiler [Kopetz and Nossal, 1995] can construct a time-triggered schedule for the communication system. At the end of the architecture design phase, the precise interface specifications of components are available, which are the inputs and constraints for the component design.

During the component design phase, the development of the host application software is performed. The precise specification of the CNI to the time-triggered communication system provides pre- and post-conditions for the application software. The host operating system can employ any reasonable scheduling strategy, as long as it ensures the timely update of the state information in the CNI in order to ensure temporal accuracy of the state information transferred to other nodes. Furthermore, in case of phase-sensitive real-time images [Kopetz, 1997], the host operating system must schedule tasks in such a way that temporal accuracy of the received state variables is guaranteed.

Event-Triggered Systems

During the design of event-triggered systems, a functional specification is mapped to an event-triggered architecture, which employs an event-triggered communication system for the establishment of the linking interface between components. Since message exchanges occur as a consequence to events, event-triggered systems offer an interrelationship between communication and computations. Hence, there is also an interrelationship between architecture and component design.

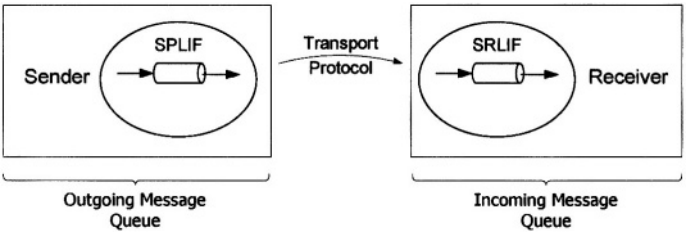

Figure 2.6. Event-Triggered Linking Interface

Compared to time-triggered systems, event-triggered systems do not require a specification of the precise global points in time of message exchanges. This weaker temporal specification enables the communication system to allow for contention and to multiplex communication resources among components. A temporal specification of the linking interfaces of components is, however, necessary to dimension buffers of an event-triggered communication system.

Figure 2.6 shows two components connected via an event-triggered linking interface. Message queues serve the purpose of buffering outgoing messages that are delayed through contention at the network, as well as incoming messages that have not yet been fetched by the application. It is not possible to dimension message queues in case of unconstrained message arrival rates, thereby leading to potential message omissions and loss of state synchronization for messages with event semantics. In order to be able to dimension outgoing queues, the temporal specification of the sender's linking interface must provide a statement about the message production rate of the sender. The production rate of the sender corresponds to the interarrival times of messages at the communication system. Equally, in order to dimension the incoming queue at the receiver, one requires a statement about the consumption rate of the receiver in relation to the sender's production rate. The consumption rate of the receiver corresponds to the service times of messages received from the communication system.

2.5 Computational Models

If large numbers of computers are to work together in computing systems, then it is necessary to have a system-wide architecture and computational model that they all support [Treleaven and Hopkins, 1981]. This section describes prevalent com-

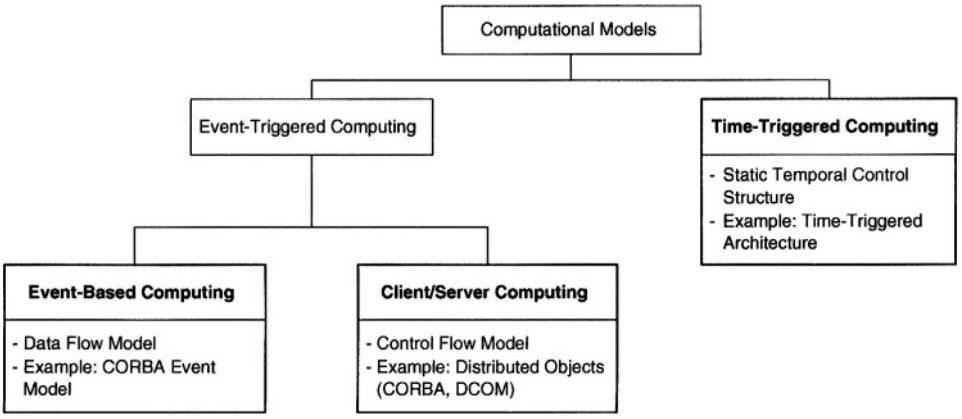

Figure 2.7. Hierarchy of Prevalent Computational Models

putational models for the representation and analysis of the design of distributed

computer systems: event-based, client/server, and time-triggered computing (see Figure 2.7). We will describe the requirements for an underlying platform (communication infrastructure, hosts) and the application areas of these computational models with an emphasis on the applicability for the construction of real-time systems.

Event-Based Computing

In *event-based models,* the glue that ties together distributed components is an event notification service (or event service). An *event notification service* is an application-independent infrastructure, whereby generators of events publish event notifications to the infrastructure and consumers of events subscribe with the infrastructure to receive relevant notifications [Carzaniga et al., 1999]. An event notification service distinguishes between the role of an event supplier and the role of event consumer. An event supplier asynchronously communicates event data to a group of event consumers as depicted in Figure 2.8. The component interactions are

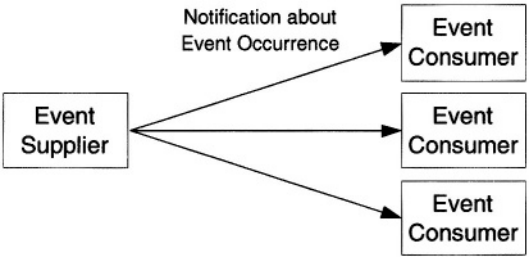

Figure 2.8. Event-Based Communication

modeled as asynchronous occurrences and responses to events. In order to inform other components about the occurrence of an internal event (a state change), components send notifications containing information about that event. Upon receiving a notification, a component can react by performing actions that, in turn, may result in the occurrence of other events [Carzaniga et al., 1999].

At the communication infrastructure, event notifications result in the sporadic transmission of event messages. Event messages carry event information and are triggered by the occurrence of the corresponding event. Hence, event-based models require an event-triggered communication service as the communication infrastructure for the transport of event notifications.

Data and Control Flow

Event-based models correspond to the *data flow model* [Veen, 1986]. The event notifications in event-based models represent a combination of control and data flow. Event information received by an event consumer represents input data that triggers computations, often in combination with other event notifications.

In the data flow model, a computation is represented by directed graphs, which illustrates the flow of data between operations. The nodes in this directed graph represent the operations, edges represent data paths. Data flow is a distributed model of computation, i.e. there is no single location of control. Furthermore, data flow is an asynchronous model that allows the execution of an operation when all input data is available. In the directed graph, the availability of inputs is indicated by a data token on each of the input arcs of a node. Data tokens represent partial results, which are consumed by the execution of an operation. Since executions use up input data tokens, these tokens are no longer available as input to other executions. The computational results of an operation are represented by releasing result tokens. The data flow model provides no concept of shared data. Precedence constraints between computational activities are expressed via the flow of data. Since an operation is activated only by the availability of input data it depends on, the overall computation is inherently parallel and asynchronous.

When building a system based on the distributed data flow model, a mapping between suppliers and consumers is required at the level of the underlying communication infrastructure. This mapping ensures that the event data of the correct supplier forms the input to a particular computational activity at a consumer. The mapping between suppliers and consumers can occur either statically or dynamically. In case of a static mapping, the different data flows are identified at design-time. A dynamic mapping allows consumers to express interest in a particular event at run-time. A dynamic mapping also enables producers to announce the availability of a particular class of events at run-time. For example, *publish-subscribe models* [Busi and Zavattaro, 2001] provide such a dynamic mapping between suppliers and consumers. In publish-subscribe models a consumer registers at a supplier (or a mediator) in order to request the delivery of future events from the supplier.

Example – CORBA Event Model

An example of a distributed event-based architecture is the Common Object Request Broker Architecture (CORBA) Event Model [OMG, 2002b] by the Object Management Group (OMG) (also called CORBA notification service). This model constructs a publish-subscribe model on top of a client/server architecture. Both suppliers and consumers are CORBA clients and use event channels that are part of the notification service for exchanging event data. Transported event data can be unstructured or structured. In the latter case, the structure is described by the OMG Interface Definition Language (IDL) [OMG, 2002a]. A single event channel can support multiple suppliers and consumers. Furthermore, event channels can consume information of other event channels, which results in the composition of event channels.

In order to limit the load resulting from events, the CORBA notification service applies filtering. Filter objects incorporate a set of constraints, which represent boolean filtering expressions. The filtering expressions are specified in a dedicated

constraint language. Filter objects can be used both at the supplier and the consumer side. They can also be composed to perform hierarchical filtering.

An application of an event-based model for real-time systems is the real-time CORBA event service described in [Harrison et al., 1997]. This work extends the CORBA notification service with support for event correlation by combining several events with conjunctive or disjunctive semantics. In addition, the real-time CORBA event service offers object-oriented event dispatching and priority scheduling. The dispatcher uses priorities for handling events from suppliers. When the dispatcher receives a supplier event, it queries the run-time scheduler to determine the priority of the consumer the event is destined for. This priority value determines the priority queue into which the event is inserted. The dispatcher supports the preemption of running threads in order to dispatch a high priority event immediately. In addition, the real-time CORBA event service provides an event channel with a non-preemptive earliest-deadline first scheduling strategy that does not use event priorities.

Client/Server Computing

In the *client/server computing paradigm,* one or more clients and one or more servers, along with the underlying operating system and interprocess communication systems, form a composite system allowing distributed computations [Sinha, 1992]. Clients send requests to a server, which processes the client requests and returns the results. Clients and servers typically run on different computers interconnected by a network [Maffeis, 1998] that is responsible for transferring requests from clients to servers and responses from servers to clients.

In real-time systems, correctness of a client/server interaction depends not only on the value of the result returned by the server, but also on the temporal behavior between the request and the corresponding result. The temporal behavior can be described with three parameters [Kopetz, 1997]: The minimum time between two successive requests by clients, the maximum response time expected by the client that issues a request, and the worst-case execution time of the server for processing the request.

Data and Control Flow

The client/server computing paradigm corresponds to the *control flow model* [Treleaven and Hopkins, 1981]. In contrast to the data flow computational models, control flow models separate the flow of data and control. This separation is beneficial, if specific control patterns are required. A system can be described by a directed graph, with nodes representing computations. An arc between two nodes denotes a control flow between these nodes. In the graph representation, control tokens must be present at the input arcs of a node for an operation to be executable. The execution of an operation releases control tokens on output arcs, thereby enabling the execution of successive operations.

Example – Distributed Objects

Distributed object computing is the application of the client/server model to object-based programming. In distributed object computing, objects are pieces of software that encapsulate an internal state and make it accessible through a well defined interface (remote procedure calling). One of the widely adopted, vendor independent distributed object computing standards is the OMG CORBA specification [OMG, 2002a].

Time-Triggered Computing

The *time-triggered model of computation* [Kopetz, 1998b] aims at the representation and analysis of the design of large hard real-time systems. The time-triggered model of computation is based on time and timeliness of real-time information. It partitions a distributed computer system into nearly autonomous subsystems with small and stable interfaces between these subsystems. A time-triggered communication system connects the interfaces. The time-triggered communication system performs a periodic exchange of state messages. Through the use of state semantics, a new version of a state message can replace the previous one in the interface.

Data and Control Flow

The interface between the host and the communication system is the temporal firewall interface [Kopetz and Nossal, 1997], which represents a data sharing interface with state information semantics. The syntactic structure of the data items in the temporal firewall and the global points in time when the data items in the temporal firewall are accessed by the time-triggered communication system are a priori specified. No control signals pass a temporal firewall, which results in the independence of the temporal behavior of the host and the time-triggered communication system.

The obligations of the host application consist of the update (producer obligation) and use (consumer obligation) of the real-time images in the temporal firewall. It is the obligation of the host application to update the state information in the temporal firewall to maintain temporal accuracy. Based on the a priori knowledge about the temporal accuracy of the real-time images in the temporal firewall, the consumer must sample the information in the temporal firewall with a sampling rate that ensures that the accessed real-time image is temporally accurate at its time of use.

Example

An example for an architecture adhering to the time-triggered model of computation is the Time-Triggered Architecture (TTA) [Kopetz and Bauer, 2003], which provides a computing infrastructure for the design and implementation of dependable distributed real-time systems. The services and design principles of the TTA are presented in Section 5.1. The TTA also provides the foundation for the inte-

grated system architecture that is introduced in this book. This integrated system architecture supports both the event-triggered and time-triggered models of computation.

2.6 Distributed System Architectures for Ultra-Dependable Systems

This section gives an overview of distributed system architectures for ultra-dependable real-time systems: Spring [Stankovic, 1990], Multicomputer Architecture for Fault-Tolerance (MAFT) [Kieckhafer et al., 1988], Integrated Modular Avionics (IMA) [ARINC 651, 1991], and BASEMENT [Hansson et al., 1997]. For each system architecture, we describe the corresponding system structure, i.e. the constituting parts and the system topology. Furthermore, we discuss the communication system that interconnects the components to the overall system and analyze the services provided at the component level, such as task scheduling. Finally, we relate each architecture to the computational models introduced in Section 2.5.

Spring

The Spring architecture [Stankovic, 1990] is designed for large, complex, distributed safety-critical real-time systems. As many real-time systems are mixed-criticality systems, Spring distinguishes computations and communication along their criticality levels. A priori guarantees are provided for safety-critical activities, while best-effort guarantees are available for activities of lower criticality.

System Structure

The Spring architecture is composed of a network of multiprocessor nodes. The constituent parts of a node are a system processor, a communications processor, one or more application processors, and an I/O subsystem.

1 **System Processor:** The system processor executes the scheduling algorithm and supports the operating system services, thus reducing operating system overhead in the application processors. In addition, the system processor handles high-priority interrupts by adapting the execution plans constructed by the scheduler.

2 **Communications Processor:** The communications processor is responsible for the exchange of information between nodes.

3 **Application Processors:** One or more application processors execute the application tasks as specified by the execution plan constructed by the scheduler in the system processor. In case a system processor fails, an application processor is reconfigured to replace the system processor.

4 **I/O Subsystem:** The I/O subsystem handles low-priority interrupts and accesses I/O devices.

Communication System

As described in [Nahum et al., 1992], the communication service of the Spring architecture enables tasks to exchange messages. Tasks may communicate with other tasks running on the same processor, with tasks on different processors in the same node, and with tasks on processors of different nodes. The same communication primitives are used, independently of the location of the communicating tasks.

Tasks communicate with each other by placing messages into ports, which is denoted as *queuing*. Task remove messages from ports by performing receive operations *(dequeuing)*. Each message can have a deadline, which denotes when the message must be delivered to the receive port. Ports are described by the following properties:

- **Capacity:** The capacity of a port determines the maximum number of messages that can be stored in the port.
- **Queuing Policy:** A port's queuing policy determines the order, in which messages are stored and retrieved.
- **Overflow Policy:** The overflow policy determines the actions (e.g., discarding the message) in case a message arrives at a port with insufficient storage capacity.
- **Task Type:** Ports are owned by tasks of a particular criticality level. The task type is employed for the scheduling of communication activities, thus preventing a task of lower criticality from affecting a task with higher criticality.
- **Semantic Type:** Spring supports three semantic types: synchronous, asynchronous, and request/reply. In synchronous communication, the sender task blocks until the receiver dequeues the message from its receive port. In asynchronous communication, senders do not wait for the dequeuing of a message at the receiver. Furthermore, a receiver does not wait for the availability of messages in asynchronous communication. Request/reply semantics support client/server interactions, i.e. a request from the client results in a corresponding reply from the server.

If tasks are located on different nodes, the network connections occur through real-time virtual circuits and real-time datagrams. *Real-time virtual circuits* are dedicated channels with guaranteed maximum latencies. For safety-critical tasks, real-time virtual circuits are pre-allocated. *Real-time datagrams* provide a best-effort communication service. These datagrams attempt to deliver messages within their deadlines without providing guarantees.

In [Teo, 1995] the Spring communication service has been extended to group communication. In synchronous multicast groups, the scheduler ensures that communicating tasks are scheduled in such a way, that messages are always ready

when a receive operation is performed. Group membership must be static for synchronous multicast groups, i.e. no tasks may join a group at run-time. Asynchronous multicast groups offer higher flexibility and scheduling is performed at run-time. Asynchronous multicast groups provide, however, only best-effort guarantees for meeting message deadlines.

Component Level

Each node in the Spring architecture runs the Spring Kernel [Stankovic and Ramamritham, 1989]. The Spring Kernel provides system primitives with bounded worst-case execution times for task management, scheduling, memory management, and intertask communication.

The Spring Kernel supports three types of tasks:

1 **Critical tasks** must meet their deadlines, otherwise a catastrophic result might occur. Resources are a priori reserved for critical tasks. The Spring architecture is based on the assumption that the overall number of critical tasks is small in relation to the overall number of tasks.

2 **Essential tasks** possess timing constraints and degrade the performance of the system, if these timing constrains are not met. For economic reasons and to improve flexibility, essential tasks do not use a pre-allocation of resources. The Spring scheduling algorithm dynamically searches for a feasible schedule that maximizes the value of executed tasks. The value of a task is its full importance value, if it completes before its deadline and a diminished value (e.g., negative value or zero), if it does not make its deadline.

3 **Non-essential tasks** execute when they do not impact critical or essential tasks.

In order to avoid unpredictable blocking, each task acquires resources before it begins and releases the resources upon completion. Tasks are characterized by precedence relationships, resource requirements, importance levels, worst-case execution times, and deadlines. Tasks with a common deadline can be grouped into a task group, thereby reducing the scheduling overhead.

The Spring scheduler is composed of four levels. At the lowest level, a dispatcher for each application processor removes tasks from scheduling queues, called system task tables. For each application processor, the system task table encodes previously established scheduling decisions that incorporate a proper task order for guaranteeing all precedence and timing constraints. The second scheduling level is a local scheduler, which uses the parameters of the current task set for deciding whether a new task or task group can be scheduled locally. The third level is a distributed scheduler that tries to find a node to execute a task or task group. The fourth level is a meta-level controller that adapts to significant changes in the environment by switching scheduling algorithms.

Supported Computational Models

The Spring architecture aims at event-triggered computing, i.e. control signals are not restricted to the progression of time. Computational and communication activities are initiated through interrupts, which represent service requests from the environment. To preserve predictability in case of indeterministic environments, Spring employs functional partitioning by separating computational resources for application software and interrupt handling. Interrupts are handled by the I/O subsystem and the system processor, thus affecting the application only indirectly via the scheduler.

MAFT

Multicomputer Architecture for Fault-Tolerance (MAFT) [Kieckhafer et al., 1988] is a distributed computer architecture for ultra-dependable systems. MAFT focuses on high reliability and high performance. The minimum performance objectives of MAFT have been derived from the requirements of flight-control systems.

System Structure

A MAFT system consists of node computers interconnected by a broadcast bus network. Each node is partitioned into two separate processors called the operations controller and the application processor. The operations controller is responsible for communication, synchronization, data voting, error detection, task scheduling, and system reconfiguration. The application processor executes the application programs.

MAFT allows to redistribute application workload to support graceful degradation of application functions as resources are lost. A task reconfiguration algorithm [Kieckhafer et al., 1988] establishes an eligibility table, which denotes for each task to which nodes the task can be assigned. The reconfiguration algorithm is implemented with three independent processes. The *Global Task Activation Process* activates or deactivates tasks, thus accounting for changes in the number of available nodes. The *Task Reallocation Process* reallocates tasks among the operating nodes. The *Task-Node Status Matching Process* prevents tasks from being executed on particular nodes.

Communication System

The communication between the operations controller and the application processor occurs through an asynchronous parallel interface. The communication between operations controllers occurs via the broadcast bus network. Four types of messages are exchanged on this network:

- **Data Messages:** A data message is broadcast by the operations controller, whenever it receives a computed data value from its own application processor.

- **Scheduling Messages:** In MAFT, precedence constraints between tasks are expressed via concurrent forks/joins and conditional branches. In order to control the distributed task scheduling, MAFT establishes a consistent view on scheduling data, i.e. the tasks that have finished execution. Byzantine agreement on scheduling data occurs through the exchange of scheduling messages that denote that an application processor has completed an application task. An interactive consistency algorithm operates in synchronized transmission rounds, during which nodes rebroadcast the scheduling data received in the previous round. The required number of rounds depends on the assumed number of simultaneous malicious faults.

- **Synchronization Messages:** The clock synchronization of MAFT employs an interactive convergence algorithm [Srikanth and Toueg, 1987]. Clock synchronization is based on the exchange of synchronization messages whose transmission implicitly denotes the local clock times. A fault-tolerant voting algorithm produces voted timestamp values for adjusting local clocks.

- **Error-Management Messages:** Error handling in MAFT is based on a penalty counting mechanism. Each node maintains a base penalty count for every other node. Furthermore, the operations controller continuously monitors the behavior of all nodes and calculates an incremental penalty count. The incremental penalty count denotes the proposed penalty assessment, based on the error detections of the current *atomic period.* At the beginning of every atomic period, each node broadcasts an error report message containing error flags, the base penalty counter, the incremental penalty counter, and an identification of suspected nodes. Each node votes on the contents of the error report messages and updates its base penalty count. If a certain threshold of the base penalty count is exceeded, the operations controllers recommend the exclusion of a node from the operating set.

Each operations controller stores a copy of all shared application data values. In addition, the operations controller performs voting of data values, transparently to application processors. Voting is performed "on-the-fly" by using a newly received data value and any previously received copies. Reasonableness checks filter out data which is outside of predefined maximum and minimum limits. MAFT supports different approximate voting algorithms, such as the "median select" algorithm and the "mean of the medial extremes" algorithm. Byzantine agreement and converging algorithms allow to maintain agreement even in case a node behaves maliciously faulty.

Component Level

For scheduling, MAFT employs a fault-tolerant variation of the deterministic priority-list algorithm. Each task possesses a unique priority number. When an

application processor becomes available, the highest priority ready task on the application processor is selected for execution.

The MAFT scheduler treats all tasks as periodic. The period of each task is a multiple of the atomic period. The boundaries between atomic periods coincide with the transmissions of synchronization messages. Non-periodic behavior is obtained through conditional branching.

Supported Computational Model

MAFT supports both the event-triggered and time-triggered paradigms for communication activities. When the application controller passes data values to its operations controller, it thereby triggers the transmission of a data message. Consequently, the transmission of data occurs event-triggered, since transmissions are within the sphere of control of the application software in the application controller.

For error handling and clock synchronization, on the other hand, periodic time-triggered disseminations of error-management messages (penalty counts, error flags) and system state messages are employed. These messages are exchanged in every atomic period.

Integrated Modular Avionics

ARINC standard 651 [ARINC 651,1991] is known as Integrated Modular Avionics (IMA) and addresses the design of architectures aimed at the separate implementation and integration of avionic applications. As depicted in Figure 2.9, the construction of an IMA architecture relies on several other ARINC standards. The services of the avionic software environment are specified by ARINC 653 [ARINC 653, 2003], which is known as APplication EXecutive (APEX). APEX provides services for partition management, process management, time management, memory management, interpartition communication, intrapartition communication, and diagnosis.

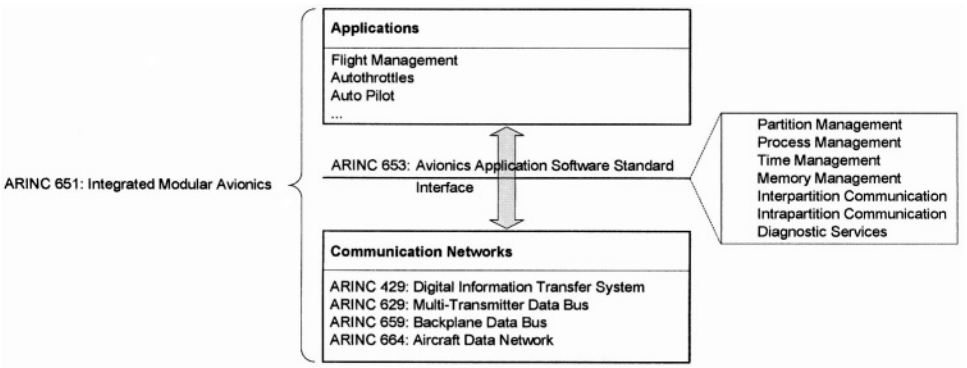

Figure 2.9. Integrated Modular Avionics

IMA systems are not restricted to a particular communication network. Common standards for the communication network include point-to-point protocols such as ARINC 429 [ARINC 429, 2001], event-triggered communication systems (e.g., ARINC 664 [ARINC 664 – Part 7, 2003], ARINC 629 [ARINC 629, 1991]), and time-triggered communication systems (e.g., ARINC 659 (SAFEbus) [ARINC 659, 1993]).

Consequently, IMA architectures are not restricted to a particular computational model. In combination with a time-triggered communication service, such as ARINC 659, an IMA architecture supports time-triggered computing with computational and communicational activities being controlled by the progression of time. In conjunction with an event-triggered communication service, such as ARINC 664, the resulting system adheres to an event-triggered computational model.

In this section, we will focus on an IMA architecture that employs the ARINC 664 implementation Avionics Full Duplex Switched Ethernet (AFDX) [ARINC 664 – Part 7, 2003]. Such an architecture allows an event-based design, in which computational and communication activities are triggered by the occurrence of significant events. The switched star network AFDX is an example of an event-triggered network complying with the ARINC 664 standard for aircraft data networks.

System Structure

With ARINC 664, the aeronautical industry explains requirements and solutions for applying standard communications protocols complying to the Open System Interconnection (OSI) Reference Model [International Standardization Organisation, 1994] for avionics applications. Consequently, ARINC 664 adopts the structuring of protocols into layers and the OSI concept of peer-to-peer communication. Each layer understands the interfaces to adjacent layers, but has no knowledge of what is occurring in other layers (see Figure 2.10).

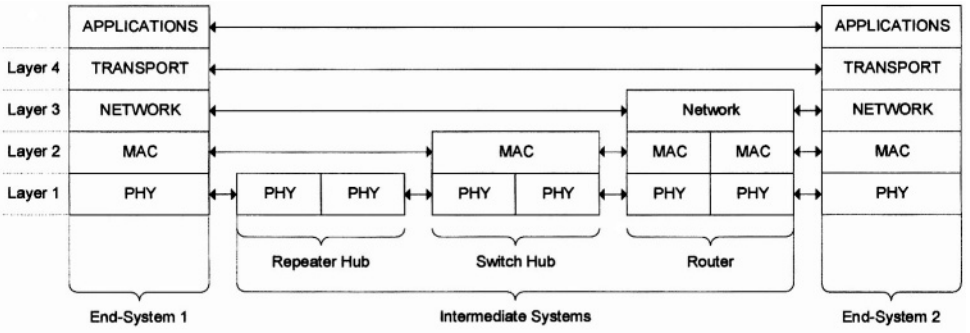

Figure 2.10. Peer-to-Peer Communications through Intermediate Systems [ARINC 664 – Part 1, 2002]

An ARINC 664 network consists of two entities: *end-systems* and *intermediate systems.* An end-system is a component that serves as a source or receptor of data

on a network, while intermediate systems implement the network. End-systems are interconnected through repeater hubs, switch hubs, and routers. A *repeater hub* forwards messages between segments. It does not perform filtering, i.e. forwarding to a segment is performed independently whether a message is destined for an end-system on the segment or not. Consequently, a repeater hub connects multiple segments into a common logical bus forming a single collision domain. A *switch hub* interconnects multiple segments into a single network at OSI layer 2. By looking at the Media Access Control (MAC) addresses (layer 2 addresses), a switch hub only relays traffic to a segment, if messages are destined for this network. A *router* operates on the Internet Protocol (IP) layer and changes the MAC destination address to the appropriate next hop, which is either an end-system or another router.

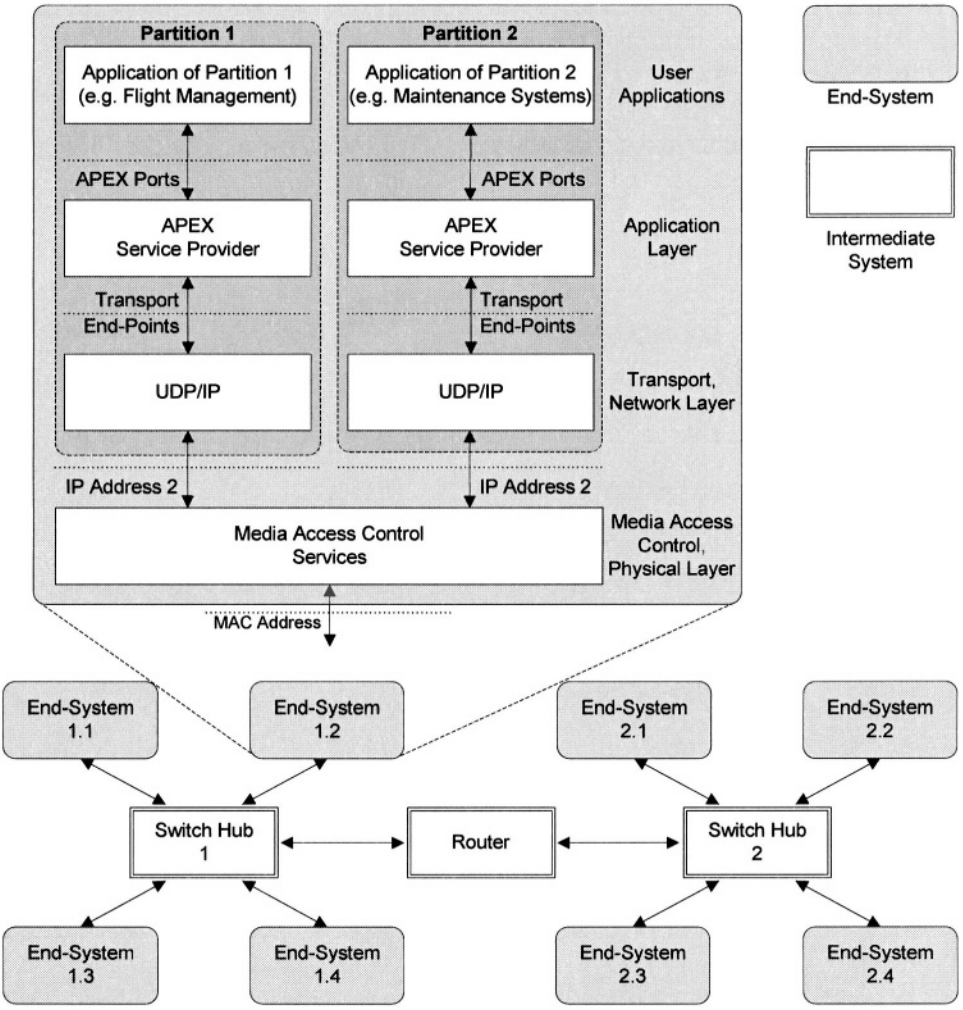

Figure 2.11. Example of an ARINC 664 Aircraft Data Network

Figure 2.11 gives an overview of an IMA system that is based on ARINC 664. The system consists of two ARINC 664 compliant subnetworks interconnected by a router. A subnetwork is composed of end-systems, each containing one or more partitions. Partitions execute application software and provide protection against the effects of software faults in other partitions. The operating environment for application software conforms to ARINC specification 653 [ARINC 653, 2003]. APEX defines the interface between the operating system and applications. Application software accesses logical communication channels via so-called APEX ports. The APEX services are part of the core software in each end-system and map the channels of APEX into end-points of the User Datagram Protocol (UDP) transport service provided by ARINC 664.

Communication System

APEX defines *channels* for interpartition communication through the exchange of messages. The destination for messages exchanged through a channel is a partition, not a process (see Figure 2.12). Communication activities are independent

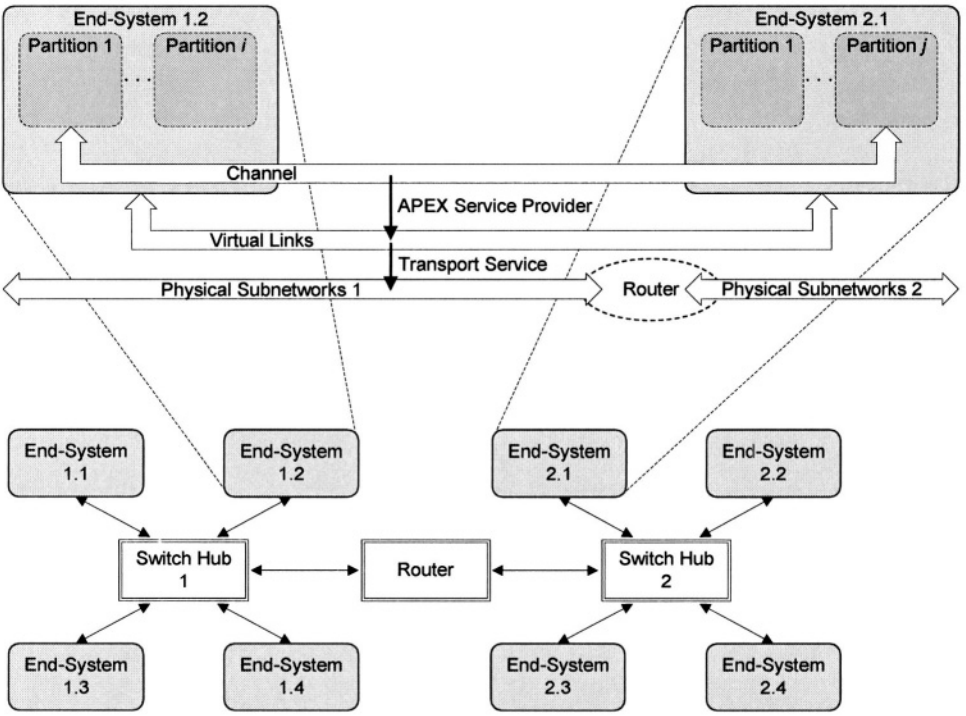

Figure 2.12. Channels and Virtual Links

of the location of both source and destination partitions. A partition can commu-

nicate via multiple channels. A channel is configured by the system integrator and possesses one sending port and one or more receiving ports. Each port is assigned a port name, which should refer to the data exchanged via the port rather than to the producer/consumer. Furthermore, a port is assigned a transfer direction (send or receive). A port can support either event or state semantics through operating in one of the following two transfer modes:

- **Sampling Mode:** Successive messages contain identical but updated data. Received messages overwrite old information, thus requiring no message queuing. Sampling mode assumes that applications are only interested in the most recent version of a message. A validity indicator denotes whether the age of the copied message is consistent with the required refresh rate defined for the port. The information semantics of messages in sampling mode corresponds to the concept of state information as introduced in Section 2.1.

  In sampling mode, a channel is assigned a single send port and one or more receive ports. Messages must possess a fixed length and arrive with a minimum frequency, as specified via the port's required refresh rate.

- **Queuing Mode:** Messages are assumed to contain uniquely different data, thus no message should be lost. The information semantics of messages in queuing mode corresponds to the concept of event information as introduced in Section 2.1. Messages are buffered in queues, which are managed on a first-in/first-out (FIFO) basis. Application software is responsible for handling overflowing queues. If a queue is full, a sending task can either cancel the send request or enter waiting state. In the latter case, the task is delayed until the queue can store the message and the task is scheduled again during the partition's time slice. Equally, a receiving task can either block when issuing a receive request on an empty queue or cancel the receive request.

  In queuing mode, a channel is assigned a single send port and a single receive port. Queuing mode permits variable message lengths.

The concept of channels, as used for interpartition communication according to APEX, is independent of the actual transport mechanism. While a time-triggered communication system simplifies the establishment of temporal validity in sampling mode, event-triggered communication systems natively support sporadic transmissions of event messages as required in queuing mode.

An example for an event-triggered avionic communication system is the ARINC 664 example implementation AFDX. AFDX is based on Ethernet and provides a dual redundant, switched star network. AFDX does not assume that transmitting end-systems are synchronized, but assumes a bounded interarrival time distribution of message transmission requests from end-systems. This assumption can be applied for limiting the effects of a faulty end-system on messages from other end-

systems. AFDX interconnects end-systems through *virtual links.* A virtual link is a communication object with one source end-system within the avionics network and one or more destination end-systems. Regardless of activities of other end-systems, a virtual link provides a guaranteed bandwidth, bounded maximum latencies and jitter, a defined probability of frame loss, and impersonation protection.

- **Guaranteed Bandwidth and Maximum Latency:** A virtual link is assigned a *bandwidth allocation gap* d_{BAG}, which is the minimum space between the first bits of two consecutive frames. The length of the bandwidth allocation gap d_{BAG} ranges between 1 ms and 128 ms with $d_{\mathrm{BAG}} = 2^k$ ms $(0 \leqslant k < 8)$. The maximum bandwidth usable by a virtual link results from its bandwidth allocation gap and the maximum frame size $L^{\max}$. The maximum usable bandwidth is $L^{\max}/d_{\mathrm{BAG}}$.
- **Maximum Jitter:** A virtual link exhibits jitter that is introduced by the switch. Jitter depends on the load and the configuration, in particular the number of virtual links. The maximum jitter is guaranteed to be less than 500 μs.
- **Defined Probability of Frame Loss:** AFDX supports redundant communication channels through a "first valid wins" policy. Transmitted Ethernet frames are assigned sequence numbers, which enable a receiver to remove frames that represent redundant copies. The first frame with a valid sequence number is accepted.
- **Impersonation Protection:** AFDX prevents an end-system from sending Ethernet frames with the source address of another end-system.

Component Level

At the component level, ARINC specification 653 distinguishes between *core software* and *application software.* The core software is responsible for mapping the APEX Application Programming Interface (API) onto the underlying platform, e.g., implementing channels through the available transport mechanisms. If the core software employs fragmentation, sequencing, routing, or message redundancy, these mechanism have to be transparent to the application software.

Each end-system contains one or more *partitions.* A partition is a set of functionally separated tasks with their associated context and configuration data. The tasks of a partition and the required resources are statically defined. The scheduling of these tasks occurs via two-level scheduling. On the first level, partitions are scheduled by the progression of time. Partitions do not have priorities and are activated relative to a time frame. As the time frame can be synchronized to the underlying communication system, the activation of partitions can also be synchronized to the communication system. Tasks within partitions possess priorities, which are used for dynamic task scheduling. If a partition is active, the scheduler selects the ready task with the highest priority for execution – preempting task's with lower priorities.

APEX distinguishes four types of software modules. The application partitions and the system partitions form the application software. The core software consists of the O/S kernel and system-specific functions.

- **Application partitions** execute software implementing application functionality. For error containment reasons, application partitions may only use ARINC 653 calls to interface the hardware and communication system.
- **System partitions** require interfaces outside of APEX services. System partitions are specific to the core software implementation.
- **The O/S kernel** provides the services defined by the APEX specification.
- **System-specific functions** implement device drivers, diagnostic, and maintenance functions.

Tasks within partitions can employ intrapartition communication mechanisms, thereby avoiding the runtime overhead of message passing. Intrapartition communication mechanisms include buffers, blackboards, semaphores, and events. Buffers and blackboards allow general communication and synchronization between tasks in a partition, while semaphores and events are used for synchronization only.

In order to protect the memory of partitions and to avoid interference in the temporal domain (e.g., through a task overrunning its deadline or blocking a shared resource), APEX demands sufficient processing, I/O and memory resources from the used processor. Furthermore, APEX requires time resources, atomic operations, and mechanisms for transferring control to the operating system, if a partition attempts to perform an invalid operation.

Supported Computational Models

IMA is not limited to a particular computational model, although the services of the underlying platform (operating system and communication system) determine the ability for supporting a computational model. For example, the availability of clock synchronization is a prerequisite for time-triggered computing.

The APEX API allows application software to explicitly request message transmissions. No distinction is made by APEX between periodic and aperiodic messages. In conjunction with an event-triggered communication system with external control, such as the ARINC 664 sample implementation AFDX, these transmission requests lead to on-demand communication activities at the network.

For handling messages with event information, APEX offers a queuing mode in order to ensure the exactly-once processing of messages. Queuing mode facilitates the establishment of state synchronization in case of messages with event information, as messages are buffered in intermediate buffers. To avoid buffer overflows, ARINC 664 proposes an error indication that is sent to transmitting applications.

APEX supports event-triggered task activations, as tasks can wait for significant events, such as the arrival of a message from the communication system. In queuing-

mode, receiver tasks can block until a message arrives in a queue, thus allowing computations to be triggered by the occurrence of events.

Sampling mode aims at the exchange of messages containing state information. A validity indicator is associated with each port operating in sampling mode. The validity indicator denotes whether the age of the copied message is consistent with the required refresh rate of a port, i.e. the temporal accuracy of the real-time image provided through the port. Sampling mode forms the basis for the development of applications conforming to the time-triggered model of computation. However, the event-triggered ARINC 664 example implementation AFDX exhibits communication jitter up to **500** μ**s**. In distributed control systems without a global time base, communication jitter leads to an uncertainty about the instant a real-time entity was observed and can be expressed as an additional error in the value domain, if there is knowledge about the maximum rate of change of the real-time entity. The communication jitter ε also limits the achievable precision of a global time base for AFDX. It is not possible to internally synchronize the clocks of an ensemble of N nodes to a better precision than $\Pi = \varepsilon \cdot \left(1 - \frac{1}{N}\right)$ [Lundelius-Welch and Lynch, 1984].

BASEMENT

In the Vehicle Internal Architecture (VIA) research project, the BASEMENT [Hansson et al., 1997] system architecture has been designed for safety-critical automotive systems. DACAPO is a realization of BASEMENT for ultra-dependable applications [Rostamzadeh et al., 1995], which has been developed in collaboration with car manufacturers. BASEMENT distinguishes between two types of applications, namely safety-critical applications and non safety-critical applications. If an application has stringent timing constraints specified via deadlines and a failure to meet such a deadline can potentially lead to an accident, the corresponding application is denoted as safety-critical. Otherwise, the application is non safety-critical.

System Structure

A BASEMENT system consists of a set of node computers interconnected with a communication network. A node computer possesses a network interface and sensors/actuators for interfacing the controlled object. In order to increase reliability, BASEMENT proposes internal redundancy of node computers and redundant networks.

Communication System

The communication system of BASEMENT divides time into slots. A subset of the available slots is statically assigned to nodes for performing the message exchanges of safety-critical applications. The allocation of slots to nodes is performed by an off-line scheduling tool [Hansson and Sjödin, 1995]. BASEMENT supports

different communication modes with corresponding precomputed communication schedules. The switching of communication modes occurs dynamically.

The remaining slots are available for non safety-critical applications and can be dynamically allocated to nodes. In the CAN-based prototype implementation of BASEMENT, contention in these communication slots is resolved by the CAN hardware arbitration mechanism.

Component Level

Tasks for safety-critical applications are periodic. The scheduling of safety-critical application tasks is controlled by a static, cyclic schedule, which is created by an off-line scheduling tool [Hansson and Sjödin, 1995]. For each node, this tools constructs a dedicated schedule that encodes all precedence requirements and statically defines the resource allocations. Computational activities are synchronized to a global time base. The length of the generated schedule is the systolic base time. At run-time, the operating system kernel ensures that the generated schedule is followed.

Tasks for non safety-critical applications are periodic or aperiodic. These tasks are scheduled by a preemptive priority driven scheduler. BASEMENT supports both dynamic and static priority assignment. No guarantees are given that deadlines of non safety-critical tasks are met.

Supported Computational Models

BASEMENT supports both the event-triggered and time-triggered computational models. Safety-critical applications employ time-triggered communication and computational activities based on a statically constructed schedule.

Non safety-critical applications can employ event-triggered communication activities with contention between nodes. Non safety-critical application tasks are usually initiated event-driven, thus reacting to the occurrence of external events.

Furthermore, the BASEMENT architecture introduces a data flow model that is based on a hardware metaphor of software. In analogy with hardware circuits, software modules are termed *software circuits* [Hansson et al., 1997]. Software circuit communication via connectors and are combined to form larger software circuits. Each software circuit possesses one or more input and output connectors. The execution of a software circuit is enabled when appropriate data is available at all input connectors. When the software circuit can perform its computations, it produces data at output connectors.For safety-critical applications, at design time software circuits are mapped to a time-triggered communication schedule by the off-line scheduling tool.

http://www.springer.com/978-0-387-23043-6

Event-Triggered and Time-Triggered Control Paradigms
Obermaisser, R.
2005, X, 153 p. 46 illus., Hardcover
ISBN: 978-0-387-23043-6