Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language

Wouter Loeve
wouterloeve0@gmail.com
University of Amsterdam
Amsterdam, Netherlands

Clemens Grelck
clemens.grelck@uni-jena.de
Friedrich Schiller University Jena
Jena, Germany

ABSTRACT

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. The functional coordination language TeamPlay follows the approach of exogenous coordination and organises an application as a streaming data-flow graph of independently operating, state-free components.

In this work we capitalise on this stringent application architecture for fault-tolerance against both permanent and transient hardware failure. We extend the TeamPlay language by a range of fault-tolerance features to be selected by the system integrator. We further propose a multi-core runtime system that is able to isolate hardware failures and manages to keep an application running flawlessly in the presence of hardware failure through adaptively morphing the application.

ACM Reference Format:
Wouter Loeve and Clemens Grelck. 2024. Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'23).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cyber-physical systems (CPS) are a computing paradigm which involves systems that interact with both the hardware they run on and the physical world around us [5]. Examples of this can be found in several areas such as self-driving cars, autonomous drones, robotics, and building & environmental control.

Commonly, these systems also have non-functional requirements. These requirements are manifested in facets such as timing, energy-consumption, security, and robustness [27, 31]. These requirements often involve trade-offs; e.g., executing an action in less time usually incurs higher power consumption. Adding robustness or fault-tolerance involves, redundancy which also incurs a higher power consumption and a higher response time.

Additionally, many safety-critical systems (especially in multi- or heterogeneous systems) require extensive testing, validation, and verification. This is often hard to do when non-functional properties, like fault-tolerance, are intertwined with computational code. In the CPS paradigm, the scientific community has been calling out to the creation of higher-level abstraction layers to alleviate the burden on the programmer [27, 31, 36].

We propose the coordination language TeamPlay [19]. Coordination languages [15] can be used to manage the interaction between separate activities or components into an often parallel system [3]. For our use-case within CPS, we envision that the coordination language can run on heterogeneous architectures thus requiring synchronisation and parallelisation [19].

The TeamPlay coordination language [19] enables the management of non-functional aspects of cyber-physical systems and facilitates the separation of concerns between coordination and computation code. Coordination code defines the structure of the application and manages non-functional properties on a high level. The computation code, in contrast, can focus on functional correctness while letting the coordination code actively manage the non-functional properties.

In this work, we make the following contributions:

(1) facilitate the management of fault-tolerance strategies in a coordination context;
(2) devise an adaptive, fault-tolerant runtime environment for TeamPlay

all while maintaining the strict separation of concerns between functional code and non-functional property management.

The remainder of the paper is organised as follows. In Section 1 we revisit the exogenous coordination model underlying the TeamPlay language before we introduce the language itself in Section 3. In Section 4 we introduce the proposed fault-tolerance extensions to the TeamPlay coordination language, and we explain our adaptive, fault-tolerant runtime environment in Section 5. At last, we sketch out related work in Section 6 before we draw conclusions and illustrate directions of current and future work in Section 7.

2 COORDINATION MODEL

The term *coordination* goes back to the seminal work of Gelernter and Carriero [16] and their coordination language Linda. Coordination languages can be classified as either *endogenous* or *exogenous* [2]. Endogenous approaches provide coordination primitives within application code; the original work on Linda falls into the category. We pursue an exogenous approach that completely separates the concerns of coordination programming and application programming. Software *components* serve as the central artefact in between.

2.1 Components

Our exogenous approach fosters the separation of concerns between intrinsic component behaviour and extrinsic component interaction. The notion of a component is the bridging point between low-level

IFL'23, August 29–31, 2023, Braga, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

functionality implementation and high-level application design. We illustrate our component model in Figure 1. Following the keyword `component` we have a unique component name that serves the dual purpose of identifying a certain application functionality and of locating the corresponding implementation in the object code.

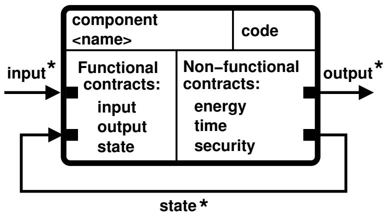


Figure 1: Illustration of component model

A component interacts with the outside world via component-specific numbers of typed and named input ports and output ports. As the Kleene star in Figure 1 suggests, a component may have zero input ports or zero output ports. A component without input ports is called a *source component*; a component without output ports is called a *sink component*. Source components and sink components form the quintessential interfaces between the physical world and the cyber-world characteristic for cyber-physical systems. They represent sensors and actors in the broadest sense. We adopt the firing rule of Petri-nets, i.e. a component is activated as soon as data (tokens) are available on each input port.

Technically, a component implementation is a function adhering to the C calling and linking conventions whose name and signature can be derived from the component specification in a defined way. This function may call other functions using the regular C calling convention. However, the execution of the function, including execution of all subsidiary functions, must not interfere with the execution environment. Exceptions to the former restriction are source and sink components that are supposed to control sensors and actors.

2.2 Stateful components

Our components are conceptually stateless. However, some sort of state is very common in cyber-physical systems. We model such state in a functionally transparent way, as illustrated in Figure 1. We employ so-called *state ports* that are short-circuited from output to input. In analogy to input ports and output ports, a component may well have no state ports, which is what we consider the norm rather than the exception.

Our approach to state is in an interesting way not dissimilar from main-stream purely functional languages, such as Haskell or Clean. They are by no means free of state either, for the simple reason that many real-world problems and phenomena are stateful. However, purely functional languages apply suitable techniques to make any state fully explicit, be it monads in Haskell [29] or uniqueness types in Clean [1]. Making state explicit is key to properly deal with state and state changes in a declarative way. In contrast, the quintessential problem of impure functional and even more so imperative languages is that state is potentially scattered all over the place. And even where this is not the case in practice, proving this property is hardly possible.

2.3 Non-functional properties

We are particularly interested in the non-functional properties of code execution, namely energy, time and security while we now add fault-tolerance for robustness against hardware failure, both permanent and transient. Hence, any component not only comes with functional contracts but additionally with non-functional contracts. These contracts can be inherently different in nature. Execution time and energy consumption depend on a concrete execution machinery. In contrast, security, more precisely algorithmic security, depends on the concrete implementation of a component, e.g. using different levels of encryption. However, different security levels almost inevitably incur different computational demands and, thus, are likely to expose different runtime behaviour in terms of time and energy consumption as well.

2.4 Multi-version components

As illustrated in Figure 2, a component may have multiple versions, each with its own energy, time and security contracts, but otherwise identical functional behaviour. More security requires stronger encryption which requires more computing and, thus, more time and energy. However, many systems do not need to operate at a maximum security level at all times. Take as an example a reconnaissance drone that adapts its security protocol in accordance with changing mission state: low security level while taking off or landing from/to base station, medium security level while navigating to/from mission area, high security level during mission.

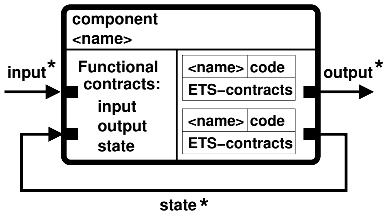


Figure 2: Multi-version component with individual energy, time and security contracts

In low security mode, the drone can use a less resilient encryption when communicating with the base station while highest possible security is paramount in a potentially hostile environment. Continuous adaptation of security levels results in less computing and, thus, in energy savings that could be exploited for longer flight times. Our solution is to embed different versions of the same component that are all functionally equivalent, but expose different trade-offs regarding non-functional properties, similar to [37].

2.5 Component interplay

Components are connected via channels to exchange data, as illustrated in Figure 3. Depending on application requirements, components may start computing at statically determined time slots, when all input data is guaranteed to be present or may be activated dynamically by the presence of all required input data. Components may produce output data on all or only on selected output ports.

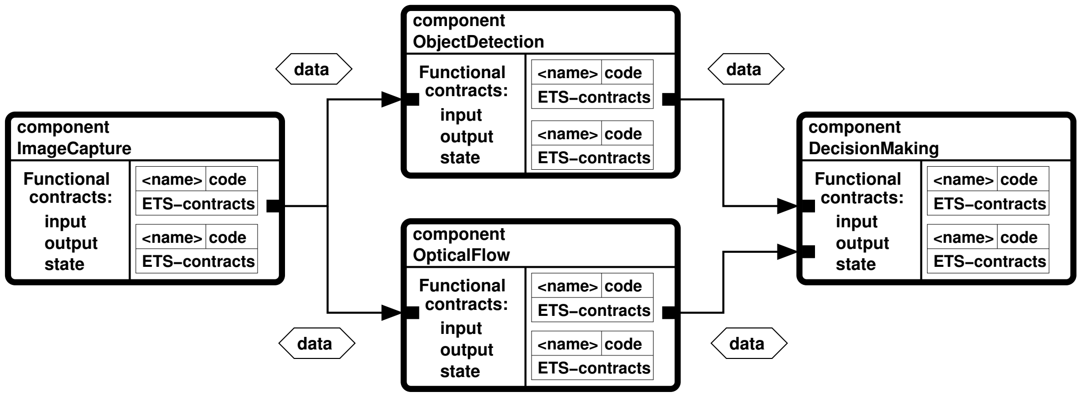


Figure 3: Illustration of data-driven component interplay via FIFO channels

3 THE TEAMPLAY LANGUAGE

We illustrate the TeamPlay coordination language by means of an example. Figure 4 shows an imaginary subsystem of a car. Two sensors feed messages to a decision controller, which synchronises the messages pair-wise and sends commands to two subsequent actuators.

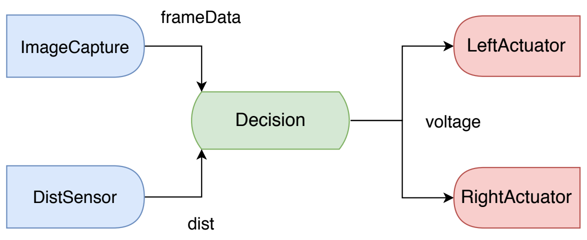


Figure 4: Example for TeamPlay component coordination

```
app car {
  components {
    DistSensor {
      outports {num dist}
    }
    ImageCapture {
      outports {frame frameData}
    }
    Decision {
      inports { num dist; frame frameData}
      outports { num voltage}
    }
    LeftActuator {
      inports {num voltage}
    }
    RightActuator {
      inports {num voltage}
  } }
  channels {
    DistSensor.dist -> Decision.dist;
    ImageCapture.frameData -> Decision.frameData;
    Decision.voltage -> LeftActuator.voltage
                      & RightActuator.voltage;
} }
```

Figure 5: TeamPlay code for example of Figure 4

Figure 5 shows the TeamPlay coordination code that implements this example. A TeamPlay application definition starts with the keyword `app` followed by an identifier that serves as the application's name. Enclosed within curly brackets we can identify the two major code regions of any TeamPLay coordination program: components with their inports and outports as well as channels connecting outports to inports.

The syntax of inport and outport specifications is inspired by C struct definitions: pairs of type name and port name separated by semicolons and embraced in curly brackets. The mapping of type names used on the coordination level and type implementations in compiled code as wel as in component implementations is not provided through the coordination program, but rather through a separate configuration file.

3.1 Components

Components serve as representations of stateless computations that map input data tokens on typed incoming streams to output data tokens emitted on typed output streams. Following the approach of exogeneous coordination the actual computation is outside the scope of the coordination code. We link our compiled coordination code with independently provided and compiled component implementation code. Given our focus on cyber-physical systems we assume component implementations to be written in C or possibly in C++.

Coming back to Figure 5 we can easily identify the definitions of the five components of our example from Figure 4. They are enclosed in curly brackets following the key word `components`. A component definition starts with a name followed by a pair of curly brackets enclosing further information about the component.

As components communicate with other components via (FIFO) channels, the corresponding ports are the most vital functional properties (or contracts) of components. Following the key words `inports`, `outports` or `state` (The latter is not shown in the running example, but the syntax is identical.) we have a list of pairs of port type and port name adopting a syntax inspired by C struct defintions. Optionally, ports can specify a multiplicity other than the default of one token, for instance as in `inports {frame in[3]}`. The multiplicity of an inport indicates the number of tokens required for firing while the multiplicity of an outport indicates the number of tokens produced on this port in a single round of firing.

Depending on their inports and outports we can classify components into three classes: *source components* with only outports, *transformer components* with both inports and outports and, at last, *sink components* with only inports. In our domain of cyber-physical applications source components control sensors while sink components control actuators.

3.2 Channels

Components are connected with each other via *channels*. In Figure 5 we can identify two kinds of channels: regular channels connect a single outport to a single inport while broadcast channels using an ampersand send a single data item from one outport to multiple inports. The TeamPlay compiler applies static analyses to guarantee type correctnes, absence of cycles and restriction that any port is connected to at most one channel. If an outport is not connected to any subsequent component, data items are systematically discarded. Input ports must always be connected to exactly one channel. Normally, ports are identified by component name and port name separated by a dot, but in the interest of concise code the port name may be omitted if a component only has a single outport or a single inport.

3.3 Non-functional properties

As pointed out before, one of the goals in the design of the TeamPlay coordination language is the active management of non-functional properties, namely energy, time and security. Both energy and time can only be considered in relation to some concrete execution machinery. Thus, any mentioning of energy or time in the coordination source code would inherently make the code hardware-specific, which is not what we want. In contrast, we employ a *non-functional properties file (NFP)* that functions as a data base storing per component time and energy consumption values for the variety of hardware execution units of interest (and perhaps even DVFS settings). Depending on the concrete hardware properties, concrete values can be derived from static code analysis, dynamic profiling or simply asserted by the user.

```
components {
  Encryption {
    inports { frame original}
    outports { frame encrypted}
    security 4;
    arch "arm/big"
}
```

Figure 6: Example of a component with non-functional properties: security and architecture

The third non-functional property of interest, security, differs from energy and time as security (in our interpretation of the word) is an algorithmic or code property that is independent of the actual execution machinery. As demonstrated in Figure 6, line 5, TeamPlay supports the specification of a security level in form of a natural number, using the `security` key word. Here, we interpret higher numbers as denoting better security. Any concrete meaning of security levels, however, are application-specific.

In a similar way we can specify an class of hardware on which the component must be scheduled for execution, using the `arch` key word and a string. This feature is relevant for the particularly targeted heterogeneous architectures, and the string refers to an architecture specification that refers to the NFP file. In the concrte example of Figure 6 the component `Encryption` is (for whatever reason) to only be scheduled on the big cores of an (imaginary) ARM big.LITTLE platform.

3.4 Multi-version components

With our focus on non-functional properties, it becomes particularly interesting to have multiple versions of a component that expose identical functional behaviour, but that implement different trade-offs of the non-functional properties of interest. Figure 7 demonstrates how this can be accomplished in TeamPlay.

```
components {
  Encryption {
    inports {frame original}
    outports {frame encrypted}
    version WeakerEncryption {security 4}
    version MediumEncryption {security 6}
    version StrongEncryption {security 9}
} }
```

Figure 7: Example of a multi-version component. The Encryption component has three different implementations, each with a different security value.

The new definition of the `Encryption` component features three different versions, distinguished by three different security levels. Different versions of one component all share the same port specifications and must behave identically from the functional perspective.

4 EXTENSIONS FOR FAULT-TOLERANCE

Our first extension of the TeamPlay language is in the specification of selected fault-tolerance methods known in literature. Some components or groups of components may be more important than others, depending on the target hardware, application domain, and other factors. We opt for a user-directed approach where the user can specify which of the predefined options to apply in different parts of the application. This is due to major challenges in having a compiler or scheduler analyse the criticality of a component in the application as a whole. Furthermore, the way fault-tolerance is implemented and achieved needs to be transparent to the programmer in order to make sure they they fit the application requirements. As in the previous section we illustrate the TeamPlay language extensions for fault-tolerance by example.

4.1 Checkpoint/restart

Checkpoint/restart lets the system return to a stable (backup) state when a fault has occurred [38, 40]. Generally, the downside of checkpoint/restart methods is the concrete state of some failing software unit is difficult to assess and, thus, in the worst case the entire process image needs to be saved at each checkpoint. That is prohibitively expensive, both in storage space and execution time.

Here the architecture of our coordination-based approach pays off. It creates a middleware layer where our system software can precisely keep copies of the arguments of an individual component invocation before giving control to the third-party provided component implementation. The stateless nature of TeamPlay components ensures that no other data affects the computation. Note here that this property remains valid even if state ports are used as described in Section 2.2. The backup copies of the argument values only need to be stored while the component is computing. As soon as it emits its output data on its outports, the backup copies can be discarded.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  checkpoint {}
}
```

Figure 8: Defaults of the checkpoint/restart specification.

The benefits of using checkpoint/restart are (potentially) threefold: implementation is straightforward, coordination between hardware components is not needed, and it only requires extra memory and copy time but no redundant active components. Figure 8 shows how checkpoint/restart can be specified on the `Decision` component from Figure 5. Currently, our specification of checkpoint/restart has no supported options, hence the empty pair or curly brackets.

4.2 Standby or primary-backup

In standby or primary-backup methods, standby components can take over the active computing component in case of failure, as illustrated in Figure 9. Initially, the output of the primary process is used. Should a fault be detected, the output of the standby component is used instead. A distinction can be made between cold, warm and hot standby which differ in the amount of synchronisation the backup components have to the active components [26]. These types can be distinguised as follows:

cold: backup component(s) are initialised but then turned off;
warm: backup component(s) are synchronised with primary component at specified points;
hot: backup component(s) are continuously synchronised with primary component.

In case of crash failures, components with long startup times benefit from this type of synchronisation because the working component can be taken over faster when using primary-backup [26]. Generally, the number of required computing resources for primary-backup are lower compared to methods like NMR. Especially using cold standby can save a lot of energy resources, which is important in (often) battery powered systems like those in the cyber-physical class. The main disadvantage of this method is that a fault needs to be detected before the redundant component can take over. Furthermore, primary-backup cannot detect value faults (i.e., there is no voting) and it relies on an error detection method to detect faults, so that the active component can be taken over. Thus, this method is primarily useful in systems which require fast switch times while keeping a state close to the state of the original, crashed process [26].

In primary-backup, the state of the primary and standby component is synchronised. The degree of this synchronisation depends

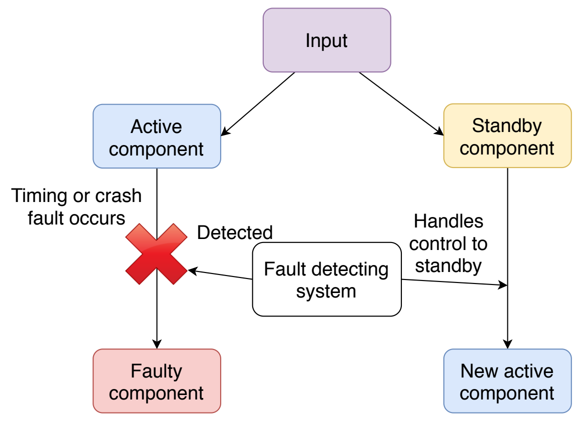


Figure 9: Illustration of recovery of a failing active component using the hot standby method. When a fault is detected, the output of the active component is routed to that of the standby component, so that the service can continue.

on which flavour, cold, warm or hot is implemented. This allows the application to quickly switch outputs when a fault is detected. In TeamPlay, the firing of a component is discrete, i.e., every execution produces output tokens only once. This, together with the fact that state is made explicit in the buffers of the edges, means that it is not necessary to run the primary and standby components at the same time, i.e., they do not have to be synchronised. We can simply provide copies of the input tokens to the hardware units and take the first unit who delivers an output as the primary component. If it fails, one of the standby components will deliver output instead. This makes this method predictable as one does not have to account for switching from the primary to the replica component or synchronisation mechanisms.

Figure 10 shows the way primary-backup can be specified. The specified options again use default values. In primary-backup the `replicas` option can be specified as an integer denoting the number of replicas to run for this component.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  standby {replicas 2}
}
```

Figure 10: Defaults of the primary-backup specification.

4.3 N-Modular redundancy

A classic example of physical redundancy is N-modular redundancy (NMR). In this strategy, n independent identical processes are executed with identical input [26, 40]. These n processes are followed by voting processes, which vote which answer they will be outputting. This method primarily focuses on masking transient faults. Depending on the fault-model for the application, it can be possible that the voter processes fail. In order to decrease the chance of this happening it is possible to increase the number of voters [4].

If a minority of the computational processes have faults, a majority vote will still result in the correct answer. Triple modular redundancy (TMR) [23, 26, 40] is a special case of NMR in which n is minimally chosen such that the computation does not have to be repeated (when a single fault is present). Since we can't know which process is likely to be faulty in case $n = 2$. However, this double-modular redundancy method has uses as an error detection method since it can be used to detect transient errors. Figure 11 illustrates N-modular redundancy with a pipeline consisting of two stages of components followed by voting processes.

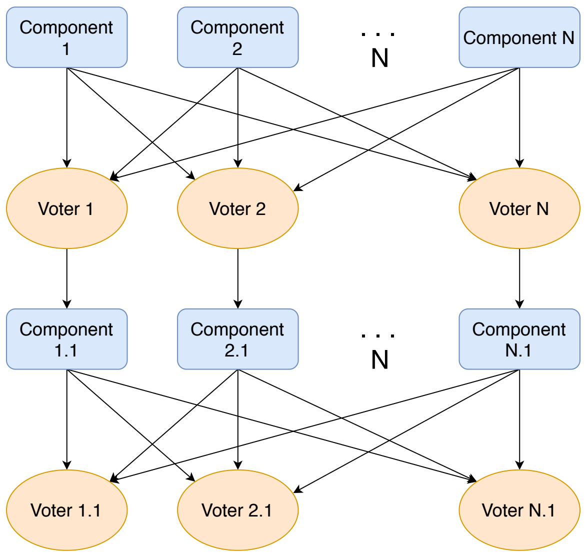


Figure 11: Illustration of N-modular redundancy

NMR is a mechanism that deals with transient faults without employing low level (hardware) error detection techniques. Furthermore, NMR is a time-predictable method, i.e., it is suitable for use in real-time systems [30]. Figure 12 shows the defaults of the N-modular redundancy. We support the following options:

- `replicas` (line 6), integer signifying the number of replicas. Default is `3`, meaning a TMR setup.
- `votingReplicas` (line 7), integer signifying whether and how much the voting processes need to be replicated.
- `waitingTime` (line 8), how long processes should wait before initiating the voting process. Given as a percentage of the average execution time of the finished components, the percentage can be higher than `100%`.
- `waitingStart` (line 9), defines the starting point of waiting. When `waitingStart` is `majority`, processes start waiting based on the execution time when a majority of processes are done. In the case of `single`, the waiting will start when a single process is ready.
- `waitingJoin` (line 10), boolean defining whether processes that are finished later should be added in the waitingTime calculation. Can apply on both a `waitingStart` value of `majority` and `single`.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  nModular {
    replicas 3
    votingReplicas 2
    waitingTime 30
    waitingStart majority
    waitingJoin true
} }
```

Figure 12: Default options for N-modular redundancy

4.4 N-version programming

In N-version programming (NVP) multiple functional equivalent implementations of the same component are created [28]. At runtime, they can be run in the same way as when using N-modular redundancy. The advantage of NVP over NMR is that software faults present in one implementation are caught the same way as transient hardware faults are caught.

The disadvantage of NVP is that it combines high runtime overhead with additional development cost. However, TeamPlay already supports the concept of multi-version components. What has primarily been intended to exploit different energy/time/security trade-offs, can now be reused for fault-tolerance. By leveraging our existing version mechanic, the programmer can kill two birds with one stone by reusing existing versions for NVP.

The options we support in NVP are similar to N-modular redundancy (Section 4.3) adding an option to specify `versions`, which defines which versions should be used and how many of each of these versions should be run. This is ilustrated in Figure 13.

```
components {
  Encryption {
    inports {frame original}
    outports {frame encrypted}
    version Encryption1 {security 4}
    version Encryption2 {security 6}
    version Encryption3 {security 9}
    nVersion {
      versions [ (Encryption1, 2) (Encryption2, 1) ]
} } }
```

Figure 13: Example of `versions`. The first entry in the tuple specifies the version while the second specifies how many replicas of that version should exist. Not all versions have to be specified because the default value is `0`.

5 FAULT-TOLERANT RUNTIME SYSTEM

In order to support the various TeamPlay language extensions specifically geared at fault-tolerance that we already described in Deliverable D1.1 we designed and implemented a corresponding fault-tolerant runtime environment. Our proof-of-concept runtime environment dynamically reconfigures running applications upon detection of hardware failures. This runtime environment targets both permanent (or crash) faults as well as transient faults, for instance detected via n-modular redundancy. The runtime environment comes with a fault injection facility for demonstration purposes.

We commence with desribing the design of the base runtime environment prior to adding fault-tolerance capabilities and its various configuration parameters. This is followed by a discussion of fault detection facilities and the concept of error tokens that aims at preventing deadlocks in the execution of TeamPlay coordinated applications. At last, we go through the various fault-tolerance techniques, as pointed out in Deliverable D1.1 and sketch out their implementations on top of our base runtime environment.

5.1 Base system

The coordination approach we take requires us to make as few assumptions as possible about the underlying target hardware configuration as our coordination approach aims to be hardware architecture agnostic. We assume that the main property of CPS(oS) holds: CPS(oS) are distributed (possibly heterogeneous) systems which are not necessarily in the same physical location [5]. This means we deal with nodes of hardware components. We simulate these distributed systems using a thread for each node with POSIX threads or *pthreads* [7].

One of the first decisions we need to make is: in what way do threads in the simulator correspond to the real world? A straightforward idea is that each coordination component corresponds with a thread. This has the advantage that it is not necessary to manage the threads separately. Since the coordination task graph is static, each component knows which thread to communicate their output data to. In other exogenous streaming coordination systems like S-Net [17] (which targets HPC systems), having components correspond with threads is feasible. But when constructing a simulator for a CPS(oS) it is not realistic to assume that there are always sufficient hardware components to accommodate each coordination component separately. Hence we choose for an architecture in which the number of computation threads is static, related to the number of hardware components in the CPS(oS) but unrelated to the number of coordination components. This requires us to work with task queues, as each thread can execute multiple components.

We choose for a design in which there are two types of threads, a main or control thread and multiple computation threads. This design is illustrated in Figure 14. The control thread checks whether components are ready and puts the tasks into their appropriate queues. It is activated as soon as a component has finished computing and matches a centralised hardware component in a real world architecture. The choice for a centralised system is motivated by security concerns as it makes it more difficult to disrupt the entire system by taking control of a single computation node. In reality, this centralised system will have to be hardened against security faults. In order to deal with faults in this management hardware component, fault-tolerance methods such as primary-backup or n-modular redundancy can be applied. The computation threads mirror the hardware components running the actual coordination component code from the real world.

5.2 Thread interaction

The interaction between the control thread and computation threads is displayed in Figure 15. The blue block on the left and green on the right indicate whether the action takes place in the worker threads or in the control thread. In the figure we show only one

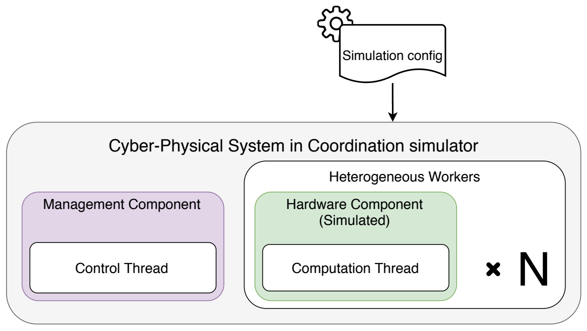


Figure 14: Illustration of the base architecture of the simulator. The management component is simulated with the use of a control thread. The heterogeneous worker elements are simulated by n computation threads.

worker thread to highlight the interaction with the control thread. The figures in the middle show the shared data structures between the control and computation threads. The top data structure is the task queue associated with the thread. The middle figure is the coordination structure containing the coordination task graph. The bottom structure is the finished list which is used to communicate that a thread is finished and tokens are added to the buffers. The tables in the figure show the first four iterations on the simple coordination structure in the middle.

First we explain how the interaction works in an abstract manner, then we go over the example listed using tables in the figure. When the control thread launches (top right node), it goes through the source nodes and adds them to the task queues of the assigned threads. Each of these threads has a counting semaphore which corresponds to the number of items in their queue. When the semaphore reaches zero, the threads wait until new items appear in the task queue.

When a thread is alerted that new items appeared in the task queue (top data structure), it pops a component from the task queue (task queues are FIFO) to execute. After execution, it stores the output data in the graph data-structure and appends the id of the executed component into the finished list. The control thread is alerted that components are finished, so it can check whether new items can be added to the task queues. The computing thread will then wait for the task queue semaphore. If the semaphore's value higher than zero it can continue popping another item from the task queue to start computing again.

After items are added to the task queues of the threads and the threads are alerted, the management thread will wait until items appear in the finished list. This is indicated in the figure by the bottom data structure with the dotted line facing right. This mechanism is implemented with a condition variable as one cannot reset a counting semaphore when the finished list is emptied. When items appear in the finished list, we need to check which components can fire again. First, we need to check whether the predecessors of the finished component can fire since, by firing, it can have opened a spot in the (bounded) FIFO buffers of the predecessors. Then, we check whether the successors of this component can fire since it has produced a token on its outports which may trigger the firing rule of the successor. Finally, we check whether the component itself can fire again. This way of checking ensures we only have to traverse the parts of the graph that have been changed. The components that can fire are added to the task queues belonging to the threads and the threads are alerted so they can continue computing. The components that are ready are added to the task queue. This marks the completion of a cycle.

Now we will explain the example found in the figure. First, both threads will launch. The worker thread sees that there are no items in the task queue (i.e., semaphore value is zero) so it will wait. In the first cycle, the control thread adds the `Source` component to the task queue. As the source component does not have any dependencies, it can fire as long as the buffers can hold the data and it is not already present in any task queue. The task is put in the task queue associated with the computing thread to which `Source` is assigned. The control thread will increment the semaphore. This leads to the awakening of the worker thread, which will pop `Source` from the FIFO queue. The worker thread will then execute the code associated with `Source` component. After computation, the output token of `Source` will be added to the buffer on the edge leading to the next component, `A`. The computing thread puts the id of the `Source` component into the finished list and sends a signal to the condition variable on which the control thread is waiting. The computing thread loops back to the first item (after initialisation) and will wait until the control thread has added new items to the task queue owned by the worker thread.

When the control thread receives the signal for the condition variable, it will loop trough the finished list and check the task graph for components which are ready. This is done by looping trough the predecessors, successors and the component itself, to see if they can fire. `Sink` has no predecessors but it does have one successor, `A`, which can fire since `Source` just fired. `Source` can also fire again. The components which can fire again are put into the task queue. Next, component `A` is fired, the result is again stored in the buffer after the fired component, this time leading to `Sink`. Then the control thread is again alerted that the worker thread has finished a computation. The control thread notices that `Source` can be fired since it has no dependencies, but it is already in a task queue, so it cannot be added again. Following the execution of `A`, `Sink` can be fired, but `A` has insufficient tokens from `Source` to fire again. Now, `Source` is taken from the task queue and executed, as it was added the previous cycle. The component checking process of this cycle is identical to the first cycle, as `Source` and `A` are added again. Then for the last round explained in this example, `Sink` will be popped from the task queue and executed. In the control thread, `A` cannot be added to the task queue again since it was already added when `Source` finished. `Sink` cannot fire again since the buffer on the edge coming from `A` does not have sufficient tokens.

5.3 Configuration file

Our simulation run-time uses a configuration file in which the user can specify options such as the number of threads and options related to fault-tolerance. Figure 16 shows an example of a configuration file. `numThreads` signifies the number of computation threads. Setting `debug` to `true` turns on debug prints. `sleepTime (cont.)` is the period of the heartbeat worker threads in microseconds. `controlSleep` is the period of the heartbeat control thread in microseconds. `heartbeatTries` is the threshold of the counter incremented by the heartbeat control thread, if the counter is higher than `heartbeatTries`, it is deemed to have crashed. `heartbeatCheckerPrio` and `heartbeatWorkerPrio` are the real-time scheduling priority of the control heartbeat thread and heartbeat worker threads respectively. Setting `standbyEarlyTaskCompletion` to `true` allows standby threads which start computation after the primary thread (i.e., first finished thread) has finished to skip computing since the task is already delivered. The `edgeBufferSize` indicates the number of tokens an edge buffer can hold.

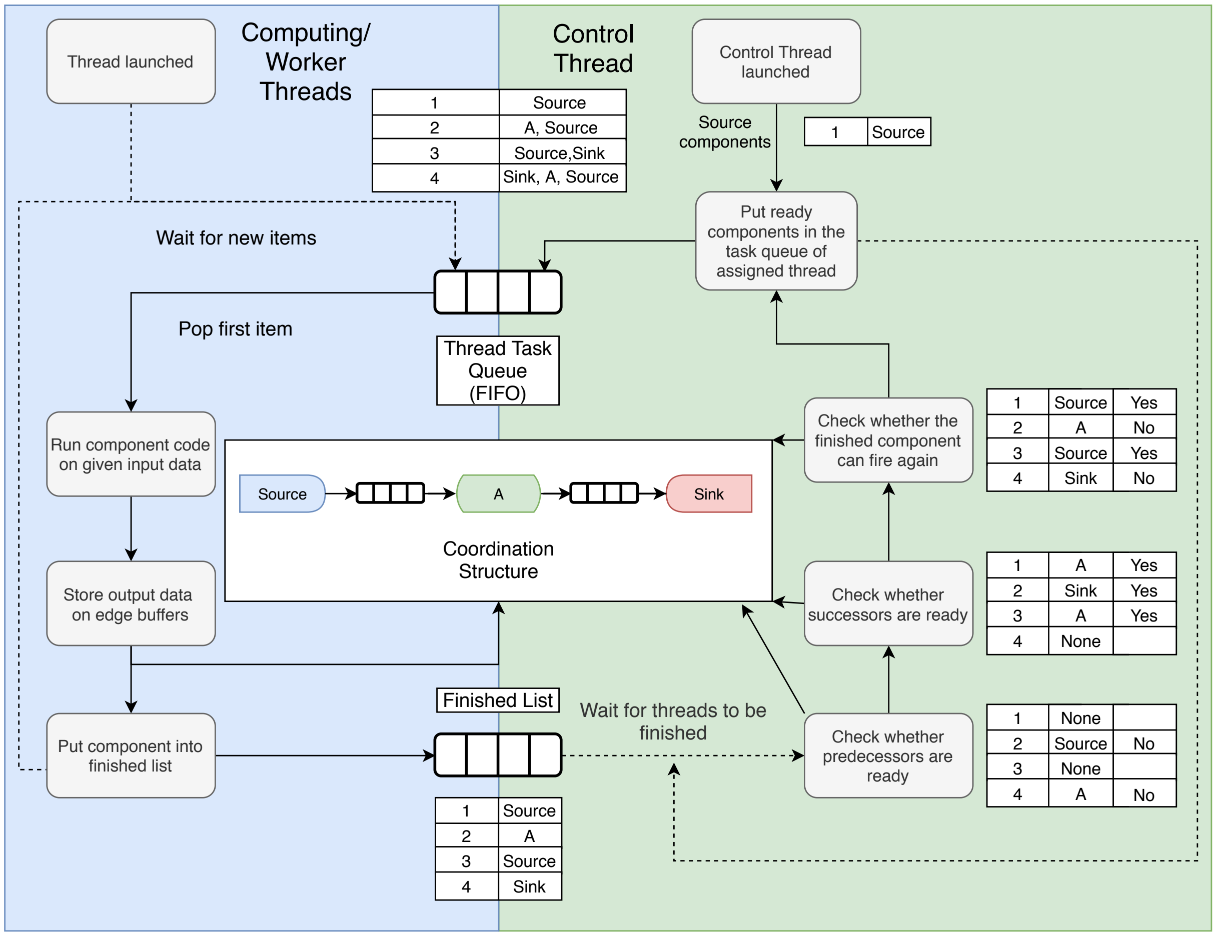
5.4 Fault-Tolerance

In order to implement the suggested fault-tolerance mechanisms, we make several additions to our base simulator. We start out by describing how we detect crash faults. This is necessary as two out of our four chosen fault-tolerance methods, checkpoint/restart and primary-backup, require error detection mechanisms that trigger these fault-tolerance methods when a fault occurs. When a system suffers from a crash failure, we cannot assume that restarting the hardware component will solve the problem. As depending on the hardware configuration, data can be stored in non-volatile storage, which is lost after the restart. To prevent deviation from correct service caused by lost data, we introduce error tokens. For the reconfiguration process, mechanisms are required which reassign tasks to other threads with a compatible architecture. Furthermore, we introduce a mechanism to deal with heterogeneous architectures and spatial assignment of components to threads.

5.5 Crash fault detection

There are three major error detection methods for detecting node failure in distributed applications [21, 40]. The first of these methods is heartbeat, in which we actively ping the nodes to know whether they have not crashed. In the second method, one passively waits for messages to come in. This method is not predictable and thus not suitable for future real-time extensions. The third method is to provide a challenge-response protocol in which the node calculates a response to a provided input. This last method can also deal with value faults in addition to detecting node failure. We choose heartbeat to deal with crash faults in our simulator as we predict this is the most predictable method with the least amount of overhead.

In order to implement heartbeat, we need a way of checking whether a computation thread has crashed. This is not straightforward, as using a signal and signal handler on an individual thread does not guarantee which thread handles the signal. Furthermore, we aim to treat the components as black boxes, so we cannot intrude into the user code to send the signal periodically from there. To solve this problem, we add a separate heartbeat thread to each computation thread. The heartbeat and a computation thread in the simulator, correspond with one hardware component in the real world. We assume that the entire (real-world) hardware component dies at once, so if the computation thread dies, the associated heartbeat thread dies with it. Additionally, we introduce a main


Computing/ Worker Threads
Control Thread
Thread launched
Control Thread launched
Source components
Wait for new items
Pop first item
Put ready components in the task queue of assigned thread
Thread Task Queue (FIFO)
Run component code on given input data
Check whether the finished component can fire again
Source
A
Sink
Coordination Structure
Store output data on edge buffers
Check whether successors are ready
Finished List
Wait for threads to be finished
Put component into finished list
Check whether predecessors are ready

1	Source
2	A, Source
3	Source,Sink
4	Sink, A, Source

1	Source

1	Source	Yes
2	A	No
3	Source	Yes
4	Sink	No

1	A	Yes
2	Sink	Yes
3	A	Yes
4	None	

1	None	
2	Source	No
3	None	
4	A	No

1	Source
2	A
3	Source
4	Sink

Figure 15: Schematic overview of the interaction between the computation components and the management component. The dotted lines indicate a waiting process via thread communication (e.g., condition variable or semaphore). The double-column tables show the components in the list per iteration. The triple-column tables show the components which are checked for firing, the third column shows whether they are added to the task queue.

```
numThreads = 6
debug = false
sleepTime = 100
controlSleep = 1000
heartbeatTries = 10
heartbeatCheckerPrio = 10
heartbeatWorkerPrio = 15
standbyEarlyTaskCompletion = false
edgeBufferSize = 20
```

Figure 16: Simulator configuration file example.

heartbeat thread or heartbeat checker thread which periodically checks whether the heartbeat threads are still alive. The checker does this by periodically looping trough the threads, incrementing a counter in memory shared with each heartbeat thread, followed by a sleep. After the increment of the counter, the heartbeat checker checks whether the counter is higher than a specified threshold, if it is higher, the worker heartbeat thread (and by extension the hardware component) is deemed unresponsive, i.e. it has crashed or is hanging. The worker heartbeat thread periodically resets the variable incremented by the checker thread, this is also followed by a sleep.

With this mechanism, we have to take care that we do not get any false positives, i.e., threads that are detected as having crashed but are still alive. Since the scheduling of the threads in the system running the simulator does not guarantee that the heartbeat worker threads wake up immediately after their sleep, we enable the user to set the amount of sleep each thread type takes as well as the threshold used by the heartbeat checker. This approach requires us to balance these three values. Choosing the worker thread sleep time too close to the heartbeat checker sleep time, will increase false-positives but will decrease the time it takes before an error is detected. Lowering the threshold has the same result, the lower the threshold, the faster errors are detected, but it also increases the chance of false-positives.

In order to further decrease the amount of false positives, we changed the scheduling type of both types of heartbeat threads to use real-time scheduling policies supported by pthreads. The function `pthread_setschedparam` supports two scheduling policies: FIFO and round robin. FIFO scheduling (`SCHED_FIFO`) runs a thread to completion in first-in-first-out order. Round robin scheduling (`SCHED_RR`) aims to give each thread an equal execution time, but involves a larger number of context switches. FIFO scheduling does not work in our system because the heartbeat threads are continuous tasks. Thus, we enabled round-robin scheduling on both types of heartbeat threads. Additionally, we enable the user to set the priority of the heartbeat checker thread and heartbeat worker threads as can be seen in Figure 16. In our case, choosing a higher priority for the heartbeat worker threads lowers the chance of having false positives, since it lowers the chance that the worker moves after the heartbeat control thread.

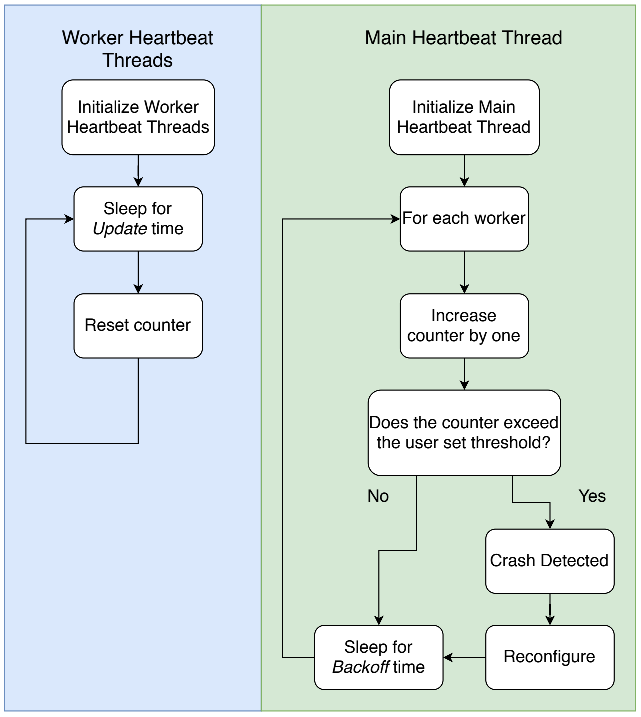


Figure 17: Overview of the heartbeat error detection mechanism. The left, blue coloured area in the image details the procedure for the worker threads while the right, green coloured area details the main heartbeat thread which checks the other threads.

5.6 Error tokens

Channels in the coordination language do not only signify data streams, but also dependency relations between components. When a thread fails and cannot be recovered, the components that follow miss tokens as the computation failed and the input tokens are lost. We explain this process and solution using the simple coordination application illustrated in Figure 18. This example contains 4 components signified by a letter. Component *Source* produces a pair of numbers and distributes them over its two outports. These outports lead to component *A* and *B* respectively. The paths from components, *A* and *B* join towards component *Sink*. What is important in the example is that the pair of numbers stay aligned, i.e. that the halves of the pair created in *Source* are consumed by the same execution of *Sink*.

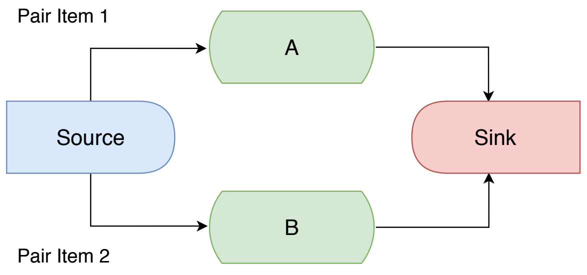


Figure 18: TeamPlay Coordination graph illustrating a branch. Component *Source* produces a pair of numbers sent to components *A* and *B* respectively. *A* and *B* are then synchronised with each other by *Sink*. If a thread executing either component *A* or *B* would crash without fault-tolerance specified one needs a mechanism to prevent misalignment to occur between the token going to the top-branch and the bottom-branch. For this we use error tokens.

When component *B* crashes and the input data is not recoverable, *B* cannot fire. This introduces a problem since component *Sink* depends on the crashed component *B*. Since *B* cannot fire, we cannot reassign it to another hardware component. This causes the pair of numbers to be misaligned, resulting in non-desired behaviour. One could argue that this kind of behaviour works for some applications, in which alignment doesn't matter but in the general case, misalignment can cause service failure. Simply skipping one firing of component *Sink* would work in this case but if there was another branch earlier in the application that leads to after component *Sink*, then this later branch would be misaligned with the result of component *Sink*.

Our solution to this problem is the introduction a fault-tolerance method, named error tokens. These error tokens indicate that one of the previous dependence relations could not be satisfied, i.e., that the following components cannot be executed without misalignment. When a thread executing a component encounters an error token it will skip the computation and produce sufficient error tokens on all outports of the component. This error token will propagate through the entire graph, invalidating the tokens that result from the same source component firing, i.e., invalidating one iteration of the application. In our example, this would mean that all numbers produced in a single firing of *Source* would be invalidated with a crash.

5.7 Checkpoint/restart

Checkpoint/restart can be implemented in the coordination language by checkpointing the FIFO buffers on the edges between the components, as the state of the entire application resides in these buffers. This is done in practice by adding an extra buffer on each outport that leads to a component. After the execution of the previous components, (i.e., the dependency components), copies of the output tokens ares made. For primitive types, this is an easy task but for user-defined types for which only a pointer is passed, the user needs to provide a copy function. When the thread executing the component fails, a new structure of input tokens is created from the checkpointed buffer and assigned to the task which takes place during the rejuvenation phase. The entire rejuvenation process, in which checkpoint/restart plays a role, is illustrated in Figure 20. We will visit this figure in full during the rejuvenation section. In normal operation, we need to remove the checkpointed data upon finishing execution and delivering the output, in order to prevent the buffers from overflowing.

5.8 Primary-backup

In primary-backup, a standby component takes over the main component when a failure is detected. Usually, this is done directly as the backup component synchronises with the active component to ensure a quick switch. In our coordination language, we do not need this behaviour as, again, the state of the application is completely saved in the FIFO buffers. We assign copies of the input tokens of the component to a number of threads equal to the number of replicas. The first thread that finishes the computation actually delivers the output. If a thread starts the computation after another thread has already delivered its answer, the thread starting the second computation can skip the task. However, it is unlikely that this behaviour is schedulable on real-time systems. Thus, we build an option into the simulator whether this form of task completion is allowed. Disabling this setting gives us the worst case, all threads compute even if the task is already delivered. When this setting is enabled, the task will only be computed multiple times if the backup threads start the computations while the thread that will finish the earliest has not yet finished. Again, the primary-backup rejuvenation process is illustrated in Figure 20.

5.9 N-version programming & N-modular redundancy

Due to time constraints we were not able to implement N-version programming and N-modular redundancy into into the simulator. We aim to explain a possible implementation of these methods into the simulator. Here we explain how N-modular redundancy can be implemented since N-version programming is a simple extension of N-modular redundancy in which you run different versions instead of identical processes.

N-modular redundancy requires more control over the processes compared to primary-backup since NMR utilises a voting process at the end. What we can use from the primary-backup system, is that these processes do not have to execute at the same time. When all processes are done or when one of the threads executing the processes has crashed or sustained a timeout conform the coordination settings, the voting process is executed on a designated voter node. This voter node requires a copy of all output tokens in order to execute the majority voting process. Our application requires that the output of NMR is a single answer, since the next component may not have NMR specified. This is why we cannot have a voter array without having an extra voting step afterwards to choose one correct answer from the voter replicas.

5.10 Component assignments & heterogeneous architectures

In order to simulate heterogeneous rejuvenation, we present an extension to the base system. Before we can create the rejuvenation mechanism we need to know which thread can be reconfigured to do which tasks. For this we need a mapping which defines which thread can simulate what component.

In time-critical systems, TeamPlay components are assigned in both the time and space dimensions to a hardware component. As the timing dimension is out of the scope of this work, we introduce a spatial assignment of the components to threads. This way, the system provides the minimum to test rejuvenation mechanisms for our fault-tolerance methods.

In addition to this, we introduce the concept of a thread class. This class mirrors a specific hardware architecture (as threads mirror hardware components) in heterogeneous systems. When a component needs to be reconfigured because it has crashed, it needs to be assigned to a component of the same class to ensure the software (e.g., component code) is compatible. This is a simplified model as in reality it is sometimes possible to run components on hardware in a different class, e.g., on a faster CPU (provided that it is compatible, as GPU code cannot run directly on a CPU). The mapping of components to threads are passed to the simulation run-time. It is not embedded in the coordination language as the number and types of threads are hardware architecture-specific.

5.11 Memory sharing

Usually, using threads on a system means that data is passed using shared memory. In cyber-physical systems (of systems) in the real world, this is generally infeasible. In our runtime, this raises the question about where the input and output data of the components should be stored. For this, we aim to solve the problem in a conceptual manner. First of all, we want to keep the strain on the internal network that connects the hardware components low. Thus, we aim to avoid sending the full content of the tokens back to the management thread for storage as it can be a CPU and network bottleneck. This is especially true with larger data-structures, e.g., images and videos, which are common in CPS(oS) [34]. This requires us to save the data on the hardware components that execute the component code and let others that need the data request it over the network.

However, this proposed solution causes problems with fault-tolerance methods like checkpoint/restart, as they require a copy of the data. This copy cannot reside on the same node, as it is not guaranteed that the data is in volatile storage and it is not guaranteed that the system can successfully restart, for example in the case of a physical failure. Our solution to this problem is the assignment of a memory-companion to each node. This memory-companion holds a copy of the checkpoint from the checkpoint/restart fault-tolerance method.

5.12 Rejuvenation

Strategies that deal with crash faults, checkpoint/restart and primary-backup require this mechanism as it is not guaranteed that crashed nodes operate normally when restarted.

Our final simulation architecture can be found in Figure 19. A hardware component from the real world is mimicked by two threads, a computation thread and a heartbeat thread. These hardware components form a group of heterogeneous workers, managed by the management component. The management component consists of two threads. The control thread checks and adds components to the task queue of the worker threads. The main heartbeat thread monitors the heartbeat threads associated with the workers and launch the rejuvenation process.

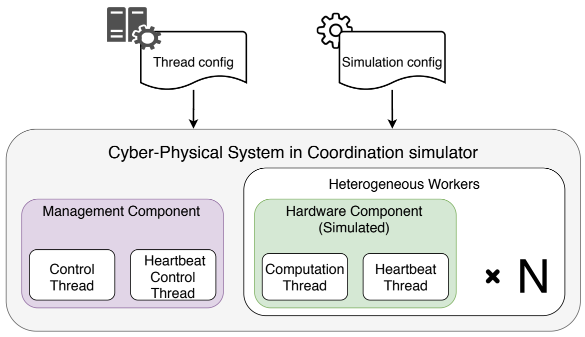


Figure 19: Illustration of the simulation architecture. The thread configuration and simulation configuration are input for the simulation runtime for the component to thread assignment and simulation settings respectively. The cyber-physical system has four thread types in two different component types. The management component consists of a control thread and a heartbeat control thread. The control thread which checks whether components are ready. The heartbeat control checks whether the heartbeat threads are still updating. The *n* heterogeneous worker threads consist of computation threads and heartbeat threads. The computation threads or worker threads execute the component code and interact with the control thread. The heartbeat thread updates a variable to show to the heartbeat control thread that it is still alive.

The rejuvenation process is illustrated in Figure 20. The process can be split into two parts: the invalidation and recovery of the task queue of the crashed thread and the reassignment of the threads' assigned components. The path of the rejuvenation process depends on which fault-tolerance mechanisms are specified. First, we explain the middle path which is taken when no fault-tolerance method is specified on the component. On this path, error tokens are produced on the outports of the components in the task queue. The component is marked as finished as the computation could not be saved by a fault-tolerance method. Then we arrive at the rejuvenation process. This rejuvenation process works by finding a non-crashed thread in the same architecture class with as extra condition that it is the thread with the fewest assigned components, to prevent from one thread taking all crashed tasks, consequently becoming a bottleneck for the application. We do not produce error tokens if a source component is present in the task queue of the crashed thread as it can simply fire again since it does not have any input tokens that need to be invalidated.

In checkpoint/restart the computation can be saved by re-running the task with the checkpointed input tokens. Before, we must reassign components to a different thread, before the checkpointed data can be rerouted there. This rejuvenation step is the same as without fault-tolerance methods except that with checkpoint/restart, tasks exist that could potentially be saved. Note that a task cannot be saved with checkpoint/restart if it crashed twice.

In primary-backup, a component is assigned to multiple threads. The number of threads depends on the number of replicas defined in the coordination language. During the invalidation of the task queue, we need to check whether a task has been delivered or not. If it has been delivered then the computation does not need to be saved or invalidated. However, if a task has not been delivered we need to check if the crashed thread is the last replica of this task. If it is the last, the computation cannot be saved and we execute the same steps as without a fault-tolerance mechanism. If there are still replicas assigned to this task we do not have to do anything since they can deliver the task as they have copies of the input data, assuming that they do not also crash. In primary-backup, we add an extra requirement for finding a new thread to which the task can be reconfigured to. This requirement is that there cannot be multiple assignments of the same component to the same task, as it defeats the purpose of primary-backup. If there are no threads left that follow these requirements the replica is not reassigned.

In our simulation, the task queue is in shared memory as the management component needs to assign new components with input data to the task queue of the thread. In a real system, you would send the task, (i.e., the component id) and where to find the input data. However, we also need to save this information on the management thread. During rejuvenation, we require a copy of this task queue in order to reassign the components of the crashed thread.

6 RELATED WORK

XBW [11] is a conceptual graphical computing model that uses entities similar to components to specify time behaviour and distribution properties. Contrary to our approach, this work makes use of uniform fault-tolerance that applies uses the same fault-tolerance techniques on the whole system, whereas TeamPlay suppports a fine-grained per component specification.

Metaobject protocols (MOPs) [13, 20] change the behaviour of object-oriented language building blocks to provide non-functional concerns, like fault-tolerance, in a systematic way. When a MOP is established, these behavioural changes result in a mostly user-transparent approach. An example of using this way of working is the FRIENDS system [12].

Fault-tolerant Linda systems [6, 13, 39] are extensions of the Linda coordination language [14, 15, 42]. This coordination language uses a tuple-space in which messages can be shared between processes. Extensions focus mostly on making tuple-space operations safer and fault-tolerant utilizing redundancy, checkpoint/restart and atomics.

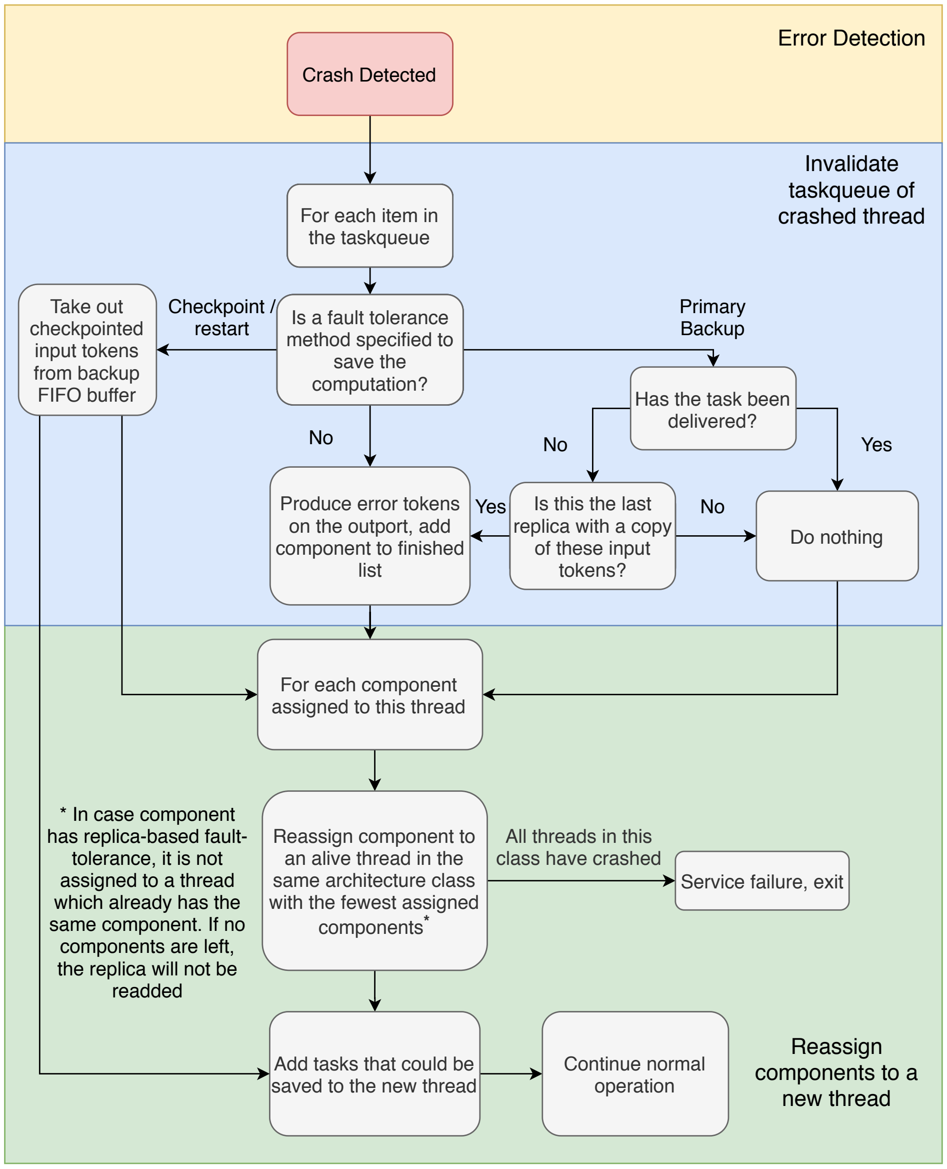

Error Detection
Crash Detected
Invalidate taskqueue of crashed thread
For each item in the taskqueue
Is a fault tolerance method specified to save the computation?
Checkpoint / restart
Take out checkpointed input tokens from backup FIFO buffer
Primary Backup
Has the task been delivered?
No
Yes
No
Produce error tokens on the outport, add component to finished list
Yes
Is this the last replica with a copy of these input tokens?
No
Do nothing
For each component assigned to this thread
* In case component has replica-based fault-tolerance, it is not assigned to a thread which already has the same component. If no components are left, the replica will not be readded
Reassign component to an alive thread in the same architecture class with the fewest assigned components*
All threads in this class have crashed
Service failure, exit
Add tasks that could be saved to the new thread
Continue normal operation
Reassign components to a new thread

Figure 20: Illustration of a crashed thread rejuvenation mechanism. The process can be split into two parts. The first part is the invalidation of the task queue of the crashed thread (the flowchart contained in the blue box up top). The second part is the reassignment of the threads assigned components which is contained in the green box below.

The Message Passing Interface (MPI) also has extensions to enable the construction of fault-tolerant programs [9, 10, 18, 22, 38]. Some extensions add structures and functions to the library, e.g., agreement algorithms and graceful error handling procedures for when nodes fail. Others focus on systematic fault-tolerance, usually with checkpoint/restart due to its low intrusiveness.

7 CONCLUSIONS

In cyber-physical systems, there is still a lot to be done in the area of creating tools, frameworks, and languages to aid the programmer in processes related to software evolution like creating and maintaining [27, 31, 36]. We have taken the first steps towards facilitating separation of concerns between computation and coordination code, thus creating the opportunity to manage non-functional properties like fault-tolerance separately from computation code.

In this work, we have extended the TeamPlay coordination language [19] by a number of fault-tolerance methods that provide fine-grained control over various forms of replication from N-modular redundancy to multi-version programming. Furthermore, we have devised a fully-fledged rutime environment that seamlessly runs TeamPlay coordination code with the specified fault-tolerance.

We are currently pursuing two directions of research. First, we plan to integrate the fault-tolerance extensions of TeamPlay as described in this paper with the time- and energy-aware heterogeneous multi-core scheduling techniques that we have developed over recent years [32, 33, 35]. Second, we work on statistical methods to quantify the impact of fault-tolerance techniques on the system reliability in the presence of single-event upsets [24, 25]. Here, we particularly address weakly-hard real-time systems, where components are permitted to fail a bounded number of times in a gliding average before disaster strikes [8].

ACKNOWLEDGEMENT

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project).

REFERENCES

[1] P. Achten and M.J. Plasmeijer. 1995. The ins and outs of Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.
[2] Farhad Arbab. 2006. Composition of Interacting Computations. In *Interactive Computation*, Dina Goldin, Scott Smolka, and Peter Wegner (Eds.). Springer, 277–321.
[3] F. Arbab, P. Ciancarini, and C. Hankin. 1998. Coordination languages for parallel programming. *Parallel Comput.* 24, 7 (1998), 989 – 1004. https://doi.org/10.1016/S0167-8191(98)00039-8
[4] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33.
[5] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *The impact of control technology* 12, 1 (2011), 161–166.
[6] D. E. Bakken and R. D. Schlichting. 1995. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems* 6, 3 (1995), 287–302. https://doi.org/10.1109/71.372777
[7] Blaise Barney. 2009. POSIX threads programming. *National Laboratory. Disponivel https://computing.llnl.gov/tutorials/pthreads* 5 (2009), 46.
[8] Guillem Bernat, Alan Burns, and Albert Liamosi. 2001. Weakly Hard Real-Time Systems. *IEEE Trans. Comput.* 50, 4 (2001), 308–321.
[9] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254.
[10] Aurélien Bouteiller. 2015. *Fault-Tolerant MPI.* Springer International Publishing, Cham, 145–228. https://doi.org/10.1007/978-3-319-20943-2_3
[11] V. Claesson, S. Poledna, and J. Soderberg. 1998. The XBW model for dependable real-time systems. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No.98TB100250).* 130–138.
[12] J. Fabre and T. Perennou. 1998. A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. *IEEE Trans. Comput.* 47, 1 (1998), 78–95.
[13] Vincenzo De Florio and Chris Blondia. 2008. A Survey of Linguistic Structures for Application-Level Fault Tolerance. *ACM Computing Survey.* 40, 2, Article 6 (May 2008), 37 pages. https://doi.org/10.1145/1348246.1348249
[14] David Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 80–112.
[15] David Gelernter and Nicholas Carriero. 1992. Coordination languages and their significance. *Commun. ACM* 35, 2 (1992), 96–108.
[16] D. Gelernter and N. Carriero. 1992. Coordination Languages and their Significance. *Commun. ACM* 35, 2 (1992), 97–107.
[17] Clemens Grelck and Frank Penczek. 2011. Implementation architecture and multithreaded runtime system of S-Net. In *Symposium on Implementation and Application of Functional Languages (IFL 2008), Lecture Notes in Computer Science 5836.* Springer, 60–79.
[18] William Gropp and Ewing Lusk. 2004. Fault Tolerance in Message Passing Interface Programs. *The International Journal of High Performance Computing Applications* 18, 3 (2004), 363–372. https://doi.org/10.1177/1094342004046045 arXiv:https://doi.org/10.1177/1094342004046045
[19] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2020. Towards Energy-, Time and Security-aware Multi-core Coordination. *22nd International Conference on Coordination Models and Languages (COORDINATION 2020), Malta* (2020).
[20] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. 1991. *The art of the metaobject protocol.* MIT press.
[21] Hermann Kopetz. 2006. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Vol. 395. Springer Science & Business Media.
[22] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. 2016. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications* 30, 3 (2016), 305–319. https://doi.org/10.1177/1094342015623623 arXiv:https://doi.org/10.1177/1094342015623623
[23] R. E. Lyons and W. Vanderkulk. 1962. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209. https://doi.org/10.1147/rd.62.0200
[24] Lukas Miedema and Clemens Grelck. 2022. Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications. In *Software Engineering and Formal Methods, 20th International Conference, SEFM 2022 (Lecture Notes in Computer Science, Vol. 13550).* Springer, 129–145.
[25] Lukas Miedema and Clemens Grelck. 2023. Change of plans: optimizing for power, reliability and timeliness for cost-conscious real-time systems. In *26th Euromicro Conference on Digital System Design (DSD 2023), Durrës, Albania.*
[26] Manuel Oriol, Thomas Gamer, Thijmen de Gooijer, Michael Wahler, and Ettore Ferranti. 2013. Fault-tolerant fault tolerance for component-based automation systems. In *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems.* 49–58.
[27] Kyung-Joon Park, Rong Zheng, and Xue Liu. 2012. Cyber-physical systems: Milestones and research challenges. *Computer Communications* 36 (2012), 1–7.
[28] Z. Peng. 2010. Building reliable embedded systems with unreliable components. In *ICSES 2010 International Conference on Signals and Electronic Circuits.* 9–13.
[29] S.L. Peyton Jones and J. Launchbury. 1995. State in Haskell. *Lisp and Symbolic Computation* 8, 4 (1995), 293–341.
[30] Stefan Poledna. 2007. *Fault-tolerant real-time systems: The problem of replica determinism.* Vol. 345. Springer Science & Business Media.
[31] R. Yogesh Rajkumar, Insup Lee, Lui Sha, and John A. Stankovic. 2010. Cyber-physical systems: The next computing revolution. *Design Automation Conference* (2010), 731–736.
[32] J. Roeder, A.D. Pimentel, and C. Grelck. 2023. GCN-based Reinforcement Learning Approach for Scheduling DAG Applications. In *Artificial Intelligence Applications and Innovations, 19th IFIP WG 12.5 International Conference, AIAI 2023, Le'on, Spain (IFIPAICT, Vol. 676).* Springer, 121–134. https://doi.org/10.1007/978-3-031-34107-6_10
[33] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2020. Energy-aware Scheduling of Multi-version Tasks on Heterogeneous Real-time Systems. In *36th ACM/SIGAPP Symposium on Applied Computing (SAC 2021).* ACM, 500–510.
[34] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2020. Towards Energy-, Time- and Security-Aware Multi-core Coordination. In *Coordination Models and Languages, 22nd International Conference, COORDINATION 2020.* Springer, LNCS 12134, 57–74.
[35] Julius Roeder, Benjamin Rouxel, and Clemens Grelck. 2021. Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems. In *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore.* IEEE.
[36] Alexander Romanovsky. 2007. A looming fault tolerance software crisis? *ACM SIGSOFT Software Engineering Notes* 32, 2 (2007), 1–4.
[37] Cosmin Rusu, Rami Melhem, and Daniel Moss'e. 2005. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing* 1, 2 (2005), 271–283.
[38] Nawrin Sultana et al. 2019. Toward a Transparent, Checkpointable Fault-Tolerant Message Passing Interface for HPC Systems. (2019).
[39] F. Tam, M. Woodward, and G. Toppong. 1995. FT-Linda: a coordination language for programming distributed fault-tolerance. In *Proceedings of IEEE Singapore International Conference on Networks and International Conference on Information Engineering '95.* 649–653. https://doi.org/10.1109/SICON.1995.526368
[40] Andrew S Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms.* Prentice-Hall.
[41] William Thies and Saman Amarasinghe. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 365–376.
[42] George Wells. 2005. Coordination languages: Back to the future with linda. In *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05).* 87–98.