Bounded-time recovery for distributed real-time systems

Neeraj Gandhi*, Edo Roth†, Robert Gifford‡, Linh Thi Xuan Phan§, and Andreas Haeberlen¶
University of Pennsylvania, Philadelphia, Pennsylvania 19104, USA
Email: {*ngandhi3, †edoroth, ‡rgif, §linhphan, ¶ahae}@seas.upenn.edu

***Abstract*—This paper explores *bounded-time recovery (BTR)*, a new approach to making cyber-physical systems robust to crash faults. Rather than trying to mask the symptoms of a fault with massive redundancy, BTR detects faults at runtime and enables the system to recover from them – e.g., by transferring tasks to other nodes that are still working correctly. When a fault does occur, there is a brief period of instability during which the system can produce incorrect outputs. However, many cyber-physical systems have physical properties – such as inertia or thermal capacity – that limit the rate at which the state of the system can change; thus, a very brief outage is often acceptable, as long as its duration can be bounded, to perhaps a few milliseconds.**

BTR has some interesting properties: for instance, it has a much lower overhead than Paxos, and, unlike Paxos, it can take useful actions even when the system partitions or a majority of the nodes fails. However, it also poses a very unusual scheduling problem that involves creating sets of interrelated schedules for different failure modes. We present a scheduling algorithm called Cascade that can quickly find suitable schedules. Using a prototype implementation, we show that Cascade scales far better than a baseline algorithm and reduces the scheduling time from hours to a few seconds, without sacrificing quality.

***Index Terms*—design space exploration for RT for latency-sensitive systems, scheduling and resource allocation for RT or latency-sensitive systems, system-level optimization and co-design techniques for RT or latency-sensitive systems**

I. Introduction

It is an unfortunate reality that nodes in a distributed system can – and frequently do – fail. This is true for all kinds of distributed systems, and cyber-physical systems (CPS) are certainly no exception: nodes can overheat, crash, lose power, malfunction, or encounter a wide range of other misfortunes. Since the failure of CPS can have dramatic consequences, including physical destruction and loss of life, it is important that they be designed to be robust to faults.

The literature contains a variety of techniques for this purpose – including, for instance, Paxos- or Raft-style fault-tolerant replication [30, 35]. However, these techniques were typically designed for general-purpose distributed systems, and not specifically for CPS. As a result, the properties they provide are not a perfect fit for CPS: on the one hand, they tend to lack properties that are critical for CPS, such as hard real-time guarantees, graceful degradation, and a low overhead; on the other hand, they also sometimes provide properties that CPS do not really need.

For instance, Paxos works very hard to create the illusion of a state machine that *never* makes a single mistake, even when many nodes fail. This makes perfect sense for, say, a banking application, where a single incorrect transaction can cause an almost unlimited amount of damage. However, it is less important for, say, an autonomous vehicle, where sending incorrect signals to the steering system is not necessarily a problem, *as long as it happens only very briefly* – e.g., for a few milliseconds. This is because the physical part of a CPS tends to have properties such as inertia or thermal capacity that limit the rate at which state changes can occur. Thus, many CPS can tolerate brief periods of incorrect actuator output [33]; the length of the period depends on the specific system, but values in the tens or hundreds of microseconds are common.

We have recently proposed a very different approach to fault tolerance that exploits this observation. Our approach, *bounded-time recovery (BTR)* [9], works by converting the system into a multi-mode system, with one mode for each anticipated fault pattern (say, "All nodes are working correctly" or "Nodes 4 and 7 have failed") and potentially a different schedule for each mode. At runtime, the nodes monitor each other to detect faults; when a fault occurs, the remaining nodes execute a transition to the appropriate mode. During the transition, the system experiences a brief period of chaos during which its outputs can be wrong; however, as long as both the time-to-detection of the fault and the transition time can be bounded, the overall system will still be able to recover quickly enough to prevent damage.

BTR has a number of interesting properties. For instance, by not trying to mask all faults perfectly, it avoids known impossibility results – such as FLP [17] – and gains the ability to take useful actions even when the system partitions or a majority of the nodes fail. It also has an inherently lower overhead than Paxos-style protocols, which generally require $2f+1$ replicas to tolerate f crash faults, whereas recovery requires only $f+1$. However, BTR introduces the major new challenge of scheduling. Since finding a new schedule within milliseconds after a fault is not realistic, it must generally prepare schedules *before* the system runs. However, preparing for up to f faults requires an undesirably large number of schedules. To make matters worse, the schedules are interdependent: if a fault can transition the system from some mode A to another mode B, then the schedules for A and B should ideally be very similar. Otherwise, the transition would require a very substantial amount of work (for shifting jobs from one node to another) and thus make it impossible to obtain a reasonable time bound. We are not aware of any scheduling problem in the literature that is even remotely similar to this.

In this paper, we focus on the question of how to approach scheduling for a BTR-type system, and, in particular how to make it scale to systems with a nontrivial number of nodes. We present an algorithm called Cascade that solves this problem in roughly the following way: first, it maps the overall scheduling problem to a very large instance of integer linear programming (ILP). Then, to obtain scalability, it recursively breaks the ILP into smaller ILPs (taking care to prevent applications from being split up across ILP instances) until it reaches a size at which solutions can be found efficiently. It then synthesizes a solution to the overall ILP, working upwards to larger and larger layers until it arrives at a schedule for the entire system. We have built a prototype implementation of Cascade, and we report results from an experimental evaluation, using both synthetic topologies and a real topology from the on-board network of a car. Our results show that Cascade can find schedules in seconds that would normally take hours to generate, with almost no loss in quality.

We note that BTR creates interesting possibilities beyond what we can explore in this paper. For instance, there is no requirement that the modes must all be variants of the original, fault-free mode. Instead, they could create new tasks (e.g., to activate an alarm signal), remove existing tasks (e.g., to triage less critical subsystems once there are no longer enough resources), or even completely change the way the system operates (e.g., to execute an emergency shutdown). This flexibility is not present in classical fault-tolerance systems, whose main goal is to *avoid* changes in the system's behavior when faults occur. In summary, our contributions in this paper are the following:

- An ILP formulation for generating schedules for BTR-style systems (Section V);
- A suitable mode transition protocol, with a matching analysis (Section VI);
- Cascade, an efficient and scalable scheduling algorithm for BTR (Section VII);
- A prototype implementation of Cascade (Section VIII);
- And an experimental evaluation (Section IX).

II. Related work

Fault tolerance: The question of how to build safe, reliable, and fault-tolerant distributed systems has been considered in great detail by several communities, including distributed systems, real-time systems, and controller design. Existing solutions include replication protocols for asynchronous distributed systems like Paxos [30] and Remus [11]; fault-tolerant real-time systems like Mars [28] and DeCoRAM [2]; fault-tolerant scheduling techniques for aperiodic real-time systems, such as deallocation and overloading [19]; and fault-tolerant and/or reconfigurable control systems [3, 4, 41]. Most of the existing work assumes crash faults, as we do in this paper, but it should be possible to extend BTR to other fault models, such as ASC [10] or Byzantine faults [31].

Self-stabilization: One way to make a distributed system fault-tolerant is to ensure that it converges to a correct state

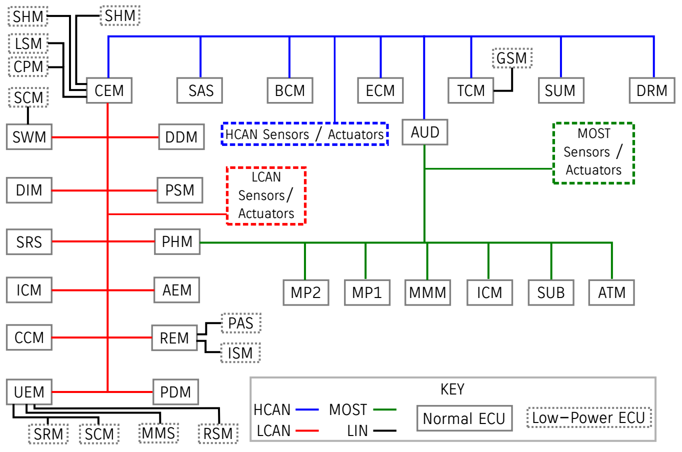

Fig. 1: Onboard network of a Volvo XC90 (based on [34]).

even if it is started in an incorrect state. This approach was first proposed by Dijkstra [13] and has led to a rich body of work on self-stabilizing systems [1, 5, 6, 14, 15, 20, 21, 24, 26, 29]. This line of work tends to use a very different system model: for instance, it often relies on a global reset mechanism for recovery, it typically does not consider scheduling, deadlines, or task criticality, and it does not provide a time bound for recovery.

Failure detectors: There is an impressive amount of work on fault detection in the context of *failure detectors*, starting from a paper by Chandra and Toueg [8], but this literature usually studies theoretical bounds on the information about failures that is necessary to solve various distributed computing problems [7]. Haeberlen et al. [23] is similar to our work in that it handles observable faults and partially connected topologies, but that work focuses on a very different setting (asynchronous system with Byzantine faults) that is mostly orthogonal to ours.

Multi-mode systems: Many real-time embedded systems can operate in multiple modes that involve different sets of tasks, and transitioning between modes requires elaborate mode-change protocols (MCPs) to prevent deadline misses and other disruptions [18, 36, 37, 38, 39, 40]. Our work is an application of the multi-mode concept to fault tolerance specifically; in the broader literature, the modes represent different conditions in which the system can operate – say, cruise control enabled or disabled – and are not necessarily aligned with failure modes. Also, the number of modes is typically much smaller, so the scalability concern does not arise.

III. System model and goals

We assume a distributed system with a number of compute nodes that execute a set of mixed-criticality applications. The nodes are connected by a network. We represent the platform by an undirected graph $G = (N, L)$, where N denotes the set of nodes and L the set of network links between two nodes.

In contrast to some of the earlier work, we *do not* assume that the network is fully connected. To see why, consider the on-board network of a Volvo XC90 – shown in Figure 1 –

which is a mix of various buses and point-to-point links. (We will use this network again in our evaluation, in Section IX-E.) We could assume *logical* connectivity, since each node can in principle reach every other node; however, in many cases there is no direct *physical* link, so messages would need to be forwarded by a few of the nodes. If these nodes fail, some of the connectivity fails with them; this can partition the network or, at the very least, increase the traffic on some of the links that would be used for "detour" routes. By modeling the physical connectivity explicitly, our solution can take these effects into account.

Each application τ_i is a dataflow of M_i tasks, $\langle \tau_{i,1} \rightarrow \tau_{i,2} \rightarrow \ldots \rightarrow \tau_{i,M_i} \rangle$, where $\tau_{i,k+1}$ relies on inputs from $\tau_{i,k}$. Each τ_i is associated with a period P_i (shared by all its tasks), a criticality C_i (smaller value means more critical), and an end-to-end deadline D_i. Each task $\tau_{i,k}$ has a worst-case execution time (WCET) of $E_{i,k}$ and a local deadline of $D_{i,k}$, and each dataflow $\tau_{i,k} \rightarrow \tau_{i,k+1}$ is associated with a latency bound of $d_{i,k}$. We assume that the deadlines $D_{i,k}$ and latency bounds $d_{i,k}$ are derived from D_i using an existing deadline decomposition technique (e.g., [12, 25, 32]), such that τ_i will always meet its end-to-end deadline if (i) all of its tasks meet their deadlines, and (ii) the latency for transmitting the output of $\tau_{i,k}$ to $\tau_{i,k+1}$ is no more than $d_{i,k}$. We assume that network links do not drop packets, and we denote by δ_s the maximum delay for transmitting a message of size s over a network link.

Fault model. We assume that up to $f < \max(|N|, |L|)$ nodes or links may fail. We assume that nodes fail by crashing, that is, they stop executing tasks and no longer send messages; this is a common model that is widely used in the fault-tolerance literature, e.g., in protocols such as Paxos [30] or Raft [35]. If a link fails, it drops all messages that are being sent over it. Depending on the network topology, a node or link failure may lead to a network partition, preventing some nodes from being able to communicate. However, in contrast to existing work, which typically assumes that the number of faults is relatively small (e.g., Paxos assumes $f < \frac{|N|}{2}$), we do not put any nontrivial restrictions on the number of faults; for instance, almost all (e.g., $|N|-1$) of the nodes could fail.

Goal. Informally, our goal is to ensure that tasks are schedulable, except for very brief periods after observable node failures. A failure on a node becomes observable [23] as soon as it affects the externally visible behavior of a node – under our fault model, this occurs when a message is omitted. More formally, a task is allowed to miss a deadline at time t iff at least one fault has become observable during $[t - R_{\max}, t]$. In other words, deadline misses are (only) allowed during intervals of length $R_{\max}$ immediately after an observable fault. We refer to these intervals as *recovery periods*, and to $R_{\max}$ as the *maximum recovery time bound*; the value of the latter depends on the specific system. We assume that $R_{\max}$ is given and applies to all applications; however, our method can be easily extended to per-application recovery times.

A major difference between existing fault-tolerance techniques and BTR is that the former try to completely mask the symptoms of faults, whereas BTR does not. In other words, classical fault tolerance will try to keep the behavior of the system constant as faults accumulate. Eventually, it runs out of resources and has to "give up" and lose all guarantees. In contrast, a BTR-enabled system can potentially change its behavior depending on the current mode; for instance, it can discard tasks to free up resources, it can launch additional tasks (e.g., an "alarm signal" task or an "emergency shutdown" task for particularly dangerous modes), or it can reconfigure itself to concentrate more resources on critical tasks, e.g., by creating additional replicas for such tasks, at the expense of less-critical ones. This provides considerable additional flexibility.

We cannot hope to fully explore the space of possible policies in this paper, so we limit ourselves to a simple default policy, which specifies, for each application τ_i, the number of additional concurrent faults $F_i(f)$ that τ_i should be able to tolerate when $f \geq 0$ faults have already occurred. For instance, the system designer could set $F_i(f) = 1$ by default and $F_i(0) = 2$ when i is a highly-critical application. This would cause the highly-critical applications to receive more replicas in the initial (fault-free) system, so they could survive a simultaneous failure of two nodes. Once a node does fail, the system would reduce the number of replicas for these applications, since there would now be fewer resources. We require that $F_i(f') \leq F_i(f)$ for all $f' > f$.

Under resource constraints, it is possible that not all applications can be scheduled. In this paper, we assume a specific scheduling objective, which is to maximize the number of applications that are schedulable (in criticality order) while meeting their fault-tolerance requirements. However, it should be possible to use our approach with other objectives.

IV. Fault detection and mode changes

Our goal is to design the system to respond to faults dynamically at runtime. Towards this, the system is designed to operate in multiple modes, each of which corresponds to a different fault scenario and may have a different mapping of tasks onto nodes. Conceptually, there is a mode for every possible failure scenario. Each mode m can be characterized by (N_m, L_m), where N_m and L_m represent the current sets of correct nodes and links; further, $f_m = |N| - |N_m|$ represents the number of nodes that have failed. The mode graph is constructed offline. At runtime, each node runs a task that detects when other nodes fail, and propagates this information to other nodes. Whenever a new fault is detected on a node, the other nodes use the recovery period to perform a mode change to the appropriate new mode. This will involve a brief period of deadline misses, as nodes transition from one mode to another, but, as long as the transition is complete before the recovery period ends, we can still maintain the bounded-time recovery guarantee.

Fault detection. Since we have assumed crash faults, we can use the following simple method to detect when they occur. (Pseudocode is included as Algorithm 1.) Each node i maintains a vector of sequence numbers, one for every node

Algorithm 1 Fault detection protocol.

```
var nodeSeq[|N|] = $(0,\dots,0)$
var linkSeq[|L|] = $(0,\dots,0)$
var currentRound = 0

function receive(sender, (inNodeSeq, inLinkSeq))
    for $i = 0\dots|N|-1$ do
        nodeSeq[i] = max(nodeSeq[i], inNodeSeq[i])
    for $i = 0\dots|L|-1$ do
        linkSeq[i] = max(linkSeq[i], inLinkSeq[i])
        if L[i] = (self, sender) or L[i] = (sender, self) then
            linkSeq[i] = currentRound

function getMode()
    var faultyNodes = $\emptyset$, faultyLinks = $\emptyset$
    for $i = 0..|N|-1$ do
        if nodeSeq[i] < (currentRound − K) then
            faultyNodes = faultyNodes $\cup$ $\{N[i]\}$
    for $i = 0..|L|-1$ do
        if linkSeq[i] < (currentRound − K) then
            faultyLinks = faultyLinks $\cup$ $\{L[i]\}$
    return (faultyNodes, faultyLinks)

function periodicUpdate()
    currentRound = currentRound + 1
    nodeSeq[self] = currentRound
    for each $j \in N : (i,j) \in L$ do
        send(j, (nodeSeq, linkSeq))
```

or link in the system; all sequence numbers are initially set to zero. Each node i also executes a special task τ_{ft}, with period P_{ft} and deadline D_{ft} that are chosen such that $D_{\text{ft}} + \delta_{\text{ft}} < P_{\text{ft}}$ where δ_{ft} is the maximum latency for transmitting the vector of sequence numbers over a network link.[1] τ_{ft} periodically increases i's sequence number in this vector and then, for each link (i,j) that is adjacent to i, sends a copy of the vector over this link (lines 23–27). When a node receives such a vector over a link (j,k), it merges it with its own vector by setting each sequence number to the maximum of its local number and the number from the vector it received (lines 5–11); additionally, it sets the sequence number for (j,k) to its own sequence number (lines 10–11). If a node i notices that its local sequence number for another node j, or a link (j,k), lags behind its own sequence number by more than a constant K (which must be greater than the diameter of the network), i concludes that the node or link has failed (lines 13–21), and transitions to the appropriate mode. Under this algorithm, the maximum detection time of a node is $\Delta_{\text{detect}} = (K+1)\cdot P_{\text{ft}}$, where P_{ft} is the period of τ_{ft}.

We make a few observations about this algorithm. First, if a node crashes or a link fails, the corresponding sequence number stops incrementing on all the other nodes, and each node can thus *independently* conclude that the other node or link is no longer working correctly. This conveniently avoids the need for a central coordinator, which could fail as well, or some form of agreement, which would require additional assumptions (such as $f < \frac{N}{2}$). Second, the mere failure of an individual link (i,j) does not cause i and j to consider each other faulty: as long as there still is *some* network path that connects i and j, updates will continue to flow along that path, although the sequence numbers will be slightly less recent than before. (If the shortest path that connects i and j has k hops, the sequence numbers will lag behind by k; hence the requirement for K to be greater than the diameter.) Third, the algorithm maintains consistency: if *any* node j is still able to hear from a node i, then all other nodes will also be able to hear from i, as long as they have a path to j. Perhaps surprisingly, this is true even when the network partitions: in this case, each partition will locally conclude that all the other nodes have failed.

[1]This requirement is to enable a simple analysis of the time-to-detection.

Replication. To provide fault tolerance, the system executes one or more replicas of a task in each mode. To avoid correlated faults, replicas are placed on different nodes. The number of replicas per application in a mode is determined based on its desirable level of fault-tolerance; specifically, for any given mode m in which f_m nodes have become faulty, we execute $F_i(f_m)+1$ replicas for each task of τ_i, so as to tolerate up to $F_i(f_m)$ additional concurrent faults. One of the replicas serves as the primary, whereas the rest are secondaries. The primary takes input from the primary of its direct predecessor (or from the sensors, if the task is the first of the dataflow), performs computation, and produces output to the replicas of its direct successor task (or to the actuators, if the task is the last one of the dataflow).

We consider two types of secondary tasks (though it should be easy to adapt our solution to other types of replication): (i) a 'hot' replica performs computation just like the primary, but discards the output; and (ii) a 'cold' replica does not perform computation; it simply accepts and logs input (obtained from the primary of its preceding task) and a checkpoint of task state (obtained from the primary). When the primary replica fails, and if the task is to be scheduled in the new mode, we appoint one of the alive secondaries to be the primary. If a hot replica becomes the new primary, it can begin producing output immediately; otherwise, the new primary will first replay the log to obtain the most recent task state before producing output.

In the following, we first present our algorithm for constructing the modes and their transitions, assuming that the mode transition time is a given parameter. We then discuss a specific mode transition protocol used in our implementation and an analysis of the mode transition time under this protocol.

V. Building the mode graph

Recall from above that the system can operate in different modes, and that each mode represents a particular fault pattern – that is, a set of faulty nodes and faulty links. Our goal is to construct a *mode graph*, whose vertexes are modes and whose edges represent possible transitions. Each mode should be annotated with a suitable schedule for that mode (which specifies the applications that are scheduled in this mode, the number of replicas per application, and a mapping of their primary and secondary tasks to the non-faulty nodes), and each edge should be annotated with a mode transition protocol that reconfigures the system from the original mode to the destination mode.

Roadmap: We begin by describing a baseline algorithm that can generate the mode graph but is not very scalable. This algorithm proceeds in three steps: first, it generates the vertexes and edges (Section V-A), then it annotates the vertexes with schedules (Section V-B), and finally, it generates transition protocols (Section VI). In Section VII, we then present a more advanced algorithm for computing the schedules that is much more scalable.

A. Mode graph construction

In principle, a system with $|N|$ nodes and $|L|$ links can have $2^{|N|+|L|}$ different modes and $2^{2(|N|+|L|)}$ mode transitions, which would quickly become unmanageable as N and L grow. However, in practice, the number of modes and transitions is much smaller, for two reasons. First, we have assumed that at most f nodes or links can fail; this reduces the number of modes to $\binom{|N|+|L|}{f}$. And second, it makes sense to rule out transitions where the destination mode contains nodes that were faulty in the origin mode – that is, in which a faulty node or link is "resurrected". With the assumption of a reliable network, the failure detector from Section IV does not make mistakes, and even if the network were not completely reliable, allowing such transitions could lead to oscillations and make the system unstable. It seems safer to exclude nodes and links permanently once a fault has been detected on them, and to rely on the operator to manually "bless" nodes and links that have been repaired and can be readmitted into the system.

With these assumptions, we obtain a directed, acyclic mode graph, whose root is the fault-free mode, and whose leaves are the modes in which the maximum number of nodes or links have failed. To construct this graph, we can (1) create an initial vertex for the fault-free mode, and then, recursively, (2) for each vertex m in the graph, enumerate all the modes m' in which one additional node or link has failed (unless this would exceed the limit of f), create a new vertex for each such mode (unless one already exists), and then add an edge from the current vertex to this vertex. We call the vertexes m' *successor modes* of m.

B. Computing the schedule for a mode

Once the skeleton of the mode graph is in place, we must generate a schedule for each vertex. At first glance, this is a very complex problem because the schedules are interdependent: if the schedule for a vertex m were completely different from the schedule of a successor mode m' of m (that is, if the two modes map almost all tasks to different nodes), the mode-change protocol for the transition $m \rightarrow m'$ would be very expensive, and would almost inevitably take a long time, thus preventing us from obtaining a low bound on the recovery time.

However, we can use the following trick to quickly find an approximate solution. Since the leaf modes are essentially unconstrained, we can start by generating schedules for them; after that, we can move up the tree and generate schedules for the predecessor modes, taking into account the schedules that have already been chosen for their successor modes. Usually, this will work well because, by definition, the predecessor modes have more available resources (in the form of nodes that are correct in the predecessor but faulty in the successor) and thus have more degrees of freedom.

Somewhat more formally, given a mode m and the schedules of its successor modes, our goal is to compute a schedule for m such that the number of applications that are scheduled, in decreasing order of criticality, is maximized, while meeting the recovery time $R_{\max}$. For this, we will use an integer linear programming (ILP) formulation, which we describe next.

Notation. For any mode m, we denote by N_m, L_m, f_m, and H_m the set of correct nodes, the set of links between the correct nodes, the number of faults that have occurred, and the shortest distance matrix, respectively. Here, $H_m(n,n')$ gives the number of network links on the shortest path from n to n'; by definition, $H_m(n,n) = 0$ for all $n \in N$, and $H_m(n,n') = +\infty$ if there exists no path connecting n and n' in m.

Recall that for each $\tau_{i,k}$, we need to execute one primary and $F_i(f_m)$ secondary replicas. We denote by $P_{i,k}$, $E_{i,k}$, $D_{i,k}$ and $U_{i,k}$ the period, WCET, deadline and density (ratio of WCET to the minimum of period and deadline) of the primary. Similarly, we denote by $P^r_{i,k}$, $E^r_{i,k}$, $D^r_{i,k}$ and $U^r_{i,k}$ the period, WCET, deadline and density of each secondary. Let $s_{i,k}$ be the size of $\tau_{i,k}$'s output data. Then, the latency for transmitting $\tau_{i,k}$'s output over a network link is bounded by $\delta_{s_{i,k}}$. Thus, $h_{i,k} = d_{i,k}/\delta_{s_{i,k}}$ captures the maximum number of network links allowed between a node executing $\tau_{i,k}$ and a node executing its successor $\tau_{i,k+1}$.

In addition to the application tasks, each node also executes the fault-detection task τ_{ft} (see Section IV). We denote by P_{ft}, E_{ft}, D_{ft} and U_{ft} the period, WCET, deadline and density of this task, respectively. For simplicity, we assume that each node schedules tasks using the Earliest Deadline First (EDF); however, it should be possible to generalize our approach to other real-time scheduling algorithms.

Variables. We use Π and $\mathcal{R}$ to represent the mappings of task replicas to nodes in the mode m. Specifically, $\Pi^n_{i,k} = 1$ ($\mathcal{R}^n_{i,k} = 1$) if the primary (a secondary) of $\tau_{i,k}$ is mapped onto node n, and $\Pi^n_{i,k} = 0$ ($\mathcal{R}^n_{i,k} = 0$) otherwise. We use $\mathcal{A}_i$ to represent whether τ_i is active in the mode ($\mathcal{A}_i = 1$) or not ($\mathcal{A}_i = 0$). For brevity, we simply write F_i in place of $F_i(f_m)$ (i.e., the number of extra concurrent faults that τ_i should tolerate in m).

Next, we describe the four constraints (C1–C4) that our ILP formulation encodes.

C1) If τ_i is active in m (i.e., $\mathcal{A}_i = 1$), there must be exactly F_i nodes that each execute a secondary replica and another node that executes the primary replica of each task of τ_i; otherwise, no task of τ_i is scheduled in m:

$$\forall 1 \le i \le M, \forall 1 \le k \le M_i : \sum_{n \in N_m} \mathcal{R}^n_{i,k} = F_i \cdot \mathcal{A}_i \ \wedge \ \sum_{n \in N_m} \Pi^n_{i,k} = \mathcal{A}_i$$

$$\forall 1 \le i \le M, \forall 1 \le k \le M_i, \forall n \in N_m : \ \Pi^n_{i,k} + \mathcal{R}^n_{i,k} \le 1$$

C2) For each active application τ_i, the number of hops between $\tau_{i,k}$'s primary and a replica of $\tau_{i,k+1}$ is at most $h_{i,k}$:

$$\forall n,n' \in N_m, \forall 1 \le i \le M, \forall 1 \le k \le M_i - 1 :$$
$$H_m(n,n') \cdot (\Pi^n_{i,k} + \Pi^{n'}_{i,k+1} + \mathcal{R}^{n'}_{i,k+1} - 1) \le h_{i,k}$$

In the above equation, if n executes $\tau_{i,k}$'s primary ($\Pi^n_{i,k} = 1$) and n' executes a replica of $\tau_{i,k+1}$ ($\Pi^{n'}_{i,k+1} + \mathcal{R}^{n'}_{i,k+1} = 1$), then the LHS becomes $H_m(n,n')$, which is the distance between $\tau_{i,k}$'s primary and a replica of $\tau_{i,k+1}$. Otherwise, $\Pi^n_{i,k} + \Pi^{n'}_{i,k+1} + \mathcal{R}^{n'}_{i,k+1} - 1 \le 0$ and the equation holds trivially.

C3) All nodes are schedulable under EDF:

$$\forall n \in N_m : \; U_{\mathsf{ft}} + \sum_{1 \le i \le M} \sum_{1 \le k \le M_i} \{U_{i,k} \cdot \Pi^n_{i,k} + U^r_{i,k} \cdot \mathcal{R}^n_{i,k}\} \le 1$$

The LHS represents the total density of all primary tasks executed on n (encoded by $U_{i,k} \cdot \Pi^n_{i,k}$) and all secondaries that are executed on n (encoded by $U^r_{i,k} \cdot \mathcal{R}^n_{i,k}$), as well as the fault-detection task τ_{ft}.

C4) For each mode transition from m to a successor mode m', we denote by $\Delta^n_{\mathsf{mc}}(m,m')$ the maximum time each node n in m' takes to transition from m to m'. (We will discuss its computation in the next section.) Since the sum of the time-to-detection Δ_{detect} and the mode transition time should be no more than the recovery time, we must have:

$$\forall n \in N_{m'} : \; \Delta^n_{\mathsf{mc}}(m,m') \le R_{\max} - \Delta_{\mathsf{detect}}$$

Objective: Our goal is to maximize the number of applications that can be feasibly scheduled in m in the criticality order (i.e., higher criticality first). This can be formulated as follows:

$$\textbf{maximize} \sum_{1 \le i \le M} \mathcal{A}_i \cdot M^{(C_{\max} - C_i + 1)}$$

where $C_{\max} = \max_{1 \le i \le M} C_i$ is the maximum criticality value of all applications. The scale factor $M^{(C_{\max} - C_i + 1)}$ has been chosen to preserve the strict preference for higher-criticality applications.

VI. Mode transition protocol and analysis

We next present a concrete mode transition protocol for Cascade. We then analyze the maximum transition time $\Delta^n_{\mathsf{mc}}(m,m')$ used in Constraint (C4) of our ILP under this protocol. Towards this, we first distinguish the different types of tasks on each node n during the transition from m to m'.

Types of tasks. Recall that Π and $\mathcal{R}$ are the mappings of primary and secondary tasks onto nodes, respectively, in mode m. Let Π' and $\mathcal{R}'$ denote the corresponding task mappings in m'. The type of a task $\tau_{i,k}$ on n can be defined as follows:

1) Old task: It is scheduled on n in m but not in m', i.e., $\Pi^n_{i,k} + \mathcal{R}^n_{i,k} = 1$ (n executes a primary/secondary replica of $\tau_{i,k}$ in m) and $\Pi'^n_{i,k} + \mathcal{R}'^n_{i,k} = 0$ (n executes no replica of $\tau_{i,k}$ in m').

2) New task: It is scheduled on n in m' but not in m, i.e., $\Pi'^n_{i,k} + \mathcal{R}'^n_{i,k} = 1$ and $\Pi^n_{i,k} + \mathcal{R}^n_{i,k} = 0$.

3) Promoted task: It is scheduled on n as a secondary in m but as the primary in m', i.e., $\mathcal{R}^n_{i,k} = 1$ and $\Pi'^n_{i,k} = 1$.

4) Demoted task: It is scheduled on n as the primary in m but as a secondary in m', i.e., $\Pi^n_{i,k} = 1$ and $\mathcal{R}'^n_{i,k} = 1$.

5) Unchanged task: It is scheduled on n in both modes and with its role preserved, i.e., either $\Pi^n_{i,k} = \Pi'^n_{i,k} = 1$ or $\mathcal{R}^n_{i,k} = \mathcal{R}'^n_{i,k} = 1$.

Mode transition protocol. When a node n detects a new fault in a current mode m, it discards all pending and currently executing jobs (and does not release new jobs of existing tasks until the transition is completed), looks up the schedule for the new mode m' (the mode that reflects the new fault pattern, including the node that just failed), and executes the mode transition protocol to switch from m to m'. Specifically, n performs the following mode transition actions:

- Terminate all old tasks immediately.
- For each new task: spawn a new primary (or secondary) replica, and copy the latest state (or the log of input and current checkpoint) from the closest existing replica that is alive, or from the existing primary if it is alive and cold replication is used. Prime the state based on the logs and checkpoint (if any).
- For each promoted task: bring its state up to date by replaying the log, and update its parameters and behavior to be the primary.
- For each demoted task: update its parameters and behavior to be a secondary.
- Resume normal execution according to the new mode.

Analysis. We next analyze the maximum transition time under the above protocol. Since the overhead for discarding jobs is negligible, we omit it here. We assume the following bounds on the network/execution overhead [2] and some extra notations:

- $\delta^{\mathsf{state}}_{i,k}$: the maximum latency to transmit the latest state of $\tau_{i,k}$ across a network link.
- $\delta^{\mathsf{log}}_{i,k}$: the maximum latency to transmit the log and checkpoint of $\tau_{i,k}$ across a network link.
- $\delta^{\mathsf{replay}}_{i,k}$: the maximum overhead to bring the state of $\tau_{i,k}$ up to date by replaying the log from the latest checkpoint.
- δ^{spawn}: the maximum overhead to spawn a new task.
- $\delta^{\mathsf{update}}_{i,k}$: the maximum overhead to update a task's parameters and functionality according to its new role.
- $\mathcal{S}_n$: the set of non-faulty nodes n' that are reachable from n in mode m' (i.e., $n' \in N_{m'}$ and $H_{m'}(n,n') < +\infty$).
- τ_{new}, τ_{promoted} and τ_{demoted}: the sets of new, promoted, and demoted tasks on n, respectively.

The transition overheads contributed by the tasks on n depend on their types, as we discuss below.

Lemma 1 (Current tasks)**.** *The total overhead of current tasks (i.e., old, unchanged, promoted and demoted tasks) on n is at most $\Delta^{\mathsf{hot}}_{\mathsf{cur}} = \sum_{\tau_{i,k} \in \tau_{\mathsf{promoted}} \cup \tau_{\mathsf{demoted}}} \delta^{\mathsf{update}}_{i,k}$ under hot replication, and at most $\Delta^{\mathsf{cold}}_{\mathsf{cur}} = \sum_{\tau_{i,k} \in \tau_{\mathsf{demoted}}} \delta^{\mathsf{update}}_{i,k} + \sum_{\tau_{i,k} \in \tau_{\mathsf{promoted}}} (\delta^{\mathsf{update}}_{i,k} + \delta^{\mathsf{replay}}_{i,k})$ under cold replication.*

Proof: Recall that old tasks are discarded immediately, and unchanged tasks preserve their states and functionality. Hence, they contribute negligible transition overhead.

[2] As common in existing work, we assume that these overhead bounds are given. Optimizing their values is an interesting future work.

Consider any promoted task $\tau_{i,k}$. Under hot replication, $\tau_{i,k}$ needs to update its parameters and functionality to its new (primary) role, which has a maximum overhead of $\delta_{i,k}^{\mathsf{update}}$. Under cold replication, $\tau_{i,k}$ additionally needs to replay its log (stored locally on n) to bring its state up to date. Hence, the overhead it contributes is at most $\delta_{i,k}^{\mathsf{replay}} + \delta_{i,k}^{\mathsf{update}}$.

Consider any demoted task $\tau_{i,k}$. Since $\tau_{i,k}$ is currently the primary on n, its state is up to date. Hence, the only overhead it contributes is the time needed to update its parameters and functionality to its new role in m', which is at most $\delta_{i,k}^{\mathsf{update}}$.

Based on the above, the overhead that all current tasks of n contribute is the total overhead of promoted and demoted tasks. Under hot replication, this overhead is bounded by $\sum_{\tau_{i,k}\in\tau_{\mathsf{promoted}}} \delta_{i,k}^{\mathsf{update}} + \sum_{\tau_{i,k}\in\tau_{\mathsf{demoted}}} \delta_{i,k}^{\mathsf{update}}$. Under cold replication, it is bounded by $\sum_{\tau_{i,k}\in\tau_{\mathsf{promoted}}} (\delta_{i,k}^{\mathsf{update}} + \delta_{i,k}^{\mathsf{replay}}) + \sum_{\tau_{i,k}\in\tau_{\mathsf{demoted}}} \delta_{i,k}^{\mathsf{update}}$. Hence, the lemma holds. ■

We next consider the new tasks on n, first under hot replication. Since such a task is not currently scheduled on n, its state must be obtained from an existing (alive) replica.

Lemma 2. *Under hot replication, the maximum delay for the states of all new tasks of n to become available on n is*

$$\Delta_{\mathsf{state}}^{\mathsf{hot}} = \max_{\tau_{i,k}\in\tau_{\mathsf{new}}} \left\{ \min_{n_0\in\mathcal{S}_n} \{ H_{m'}(n,n_0)\cdot\delta_{i,k}^{\mathsf{state}} \mid \Pi_{i,k}^{n_0} + \mathcal{R}_{i,k}^{n_0} = 1\} \right\}.$$

Proof: Under hot replication, for each new task $\tau_{i,k}$, n obtains the latest task state from the nearest non-faulty node that executes a replica of $\tau_{i,k}$ in the current mode m. Recall that $H_{m'}(n,n_0)$ denotes the minimum number of network links between n and a connected node n_0 in mode m' (i.e., $n_0 \in \mathcal{S}_n$). Since the maximum time to transmit the task state of $\tau_{i,k}$ over one network link is $\delta_{i,k}^{\mathsf{state}}$, the time to transmit the state of $\tau_{i,k}$ from n_0 to n is at most $H_{m'}(n,n_0)\cdot\delta_{i,k}^{\mathsf{state}}$. Further, for n_0 to execute a (primary or secondary) replica of $\tau_{i,k}$ in the current mode m, we must have $\Pi_{i,k}^{n_0} + \mathcal{R}_{i,k}^{n_0} = 1$. Hence, the maximum delay for the task state of $\tau_{i,k}$ to be available on n is the minimum of $H_{m'}(n,n_0)\cdot\delta_{i,k}^{\mathsf{state}}$, for all $n_0 \in \mathcal{S}_n$ such that $\Pi_{i,k}^{n_0} + \mathcal{R}_{i,k}^{n_0} = 1$. By taking the maximum of this delay across all new tasks $\tau_{i,k}$ on n, we obtain the maximum delay for their states to be available on n. This proves the lemma. ■

Theorem 3 (Hot replication)**.** *Under hot replication, the maximum time for n to transition from m to m′ is given by*

$$\Delta_{\mathsf{mc}}^{n}(m,m') = \max\{\Delta_{\mathsf{cur}}^{\mathsf{hot}},\, \Delta_{\mathsf{state}}^{\mathsf{hot}}\} + \sum_{\tau_{i,k}\in\tau_{\mathsf{new}}} \delta^{\mathsf{spawn}} \qquad (1)$$

Proof: Under hot replication, the overhead contributed by the new tasks on n consists of the overhead for transmitting their task states from existing replicas (over the network) and the overhead for spawning the new tasks (on node n). Since communication and computation can be done in parallel, the state transfer takes place concurrently with the mode change actions for existing tasks on n. Hence, from Lemma 1 and Lemma 2, the maximum time (from the instant the fault is detected) for n to both complete the mode change actions of old, unchanged, promoted and demoted tasks and to obtain the new tasks' states is $\max\{\Delta_{\mathsf{cur}}^{\mathsf{hot}},\, \Delta_{\mathsf{state}}^{\mathsf{hot}}\}$. The maximum transition time $\Delta_{\mathsf{mc}}^{n}(m,m')$ of n can thus be obtained by adding the overhead for spawning the new tasks on n to this delay, the result of which is equal to the RHS of Equation (1). ■

We next analyze the transition time under cold replication. For this, let τ_{new}^{1} be the set of new tasks of n whose current primary replicas are still alive, and $\tau_{\mathsf{new}}^{2} = \tau_{\mathsf{new}} \setminus \tau_{\mathsf{new}}^{1}$. Under cold replication, the state of each new task $\tau_{i,k}$ in τ_{new}^{1} can be obtained from its current primary. In contrast, the state of each $\tau_{i,k}$ in τ_{new}^{2} is obtained by first transferring the log and checkpoint for $\tau_{i,k}$ from the nearest alive secondary replica, and then replaying the log. The next lemma establishes the state/log transmission time.

Lemma 4. *Under cold replication, the maximum delay to transfer the states of all tasks in τ_{new}^{1} to n is*

$$\Delta_{\mathsf{state}}^{\mathsf{cold}} = \max_{\tau_{i,k}\in\tau_{\mathsf{new}}^{1}} \left\{ \min_{n_0\in\mathcal{S}_n \wedge \Pi_{i,k}^{n_0}} H_{m'}(n,n_0)\cdot\delta_{i,k}^{\mathsf{state}} \right\}.$$

Further, the maximum delay to transfer the logs and checkpoints of all tasks in τ_{new}^{2} to n is

$$\Delta_{\mathsf{log}}^{\mathsf{cold}} = \max_{\tau_{i,k}\in\tau_{\mathsf{new}}^{2}} \left\{ \min_{n_0\in\mathcal{S}_n \wedge R_{i,k}^{n_0}} H_{m'}(n,n_0)\cdot\delta_{i,k}^{\mathsf{log}} \right\}.$$

Observe that a node n_0 currently executes the primary of $\tau_{i,k}$ *iff* $\Pi_{i,k}^{n_0} = 1$, and that it currently executes a secondary replica of $\tau_{i,k}$ *iff* $R_{i,k}^{n_0} = 1$. In addition, the maximum latency to transmit the log and checkpoint of $\tau_{i,k}$ over a network link is bounded by $\delta_{i,k}^{\mathsf{log}}$. Using these conditions, the lemma can be proven in a similar manner as Lemma 2. Due to space constraints, we omit the details.

Theorem 5 (Cold replication)**.** *Under cold replication, the maximum time for n to transition from m to m′ is given by*

$$\Delta_{\mathsf{mc}}^{n}(m,m') = \max\left\{\Delta_{\mathsf{state}}^{\mathsf{cold}},\, \max\{\Delta_{\mathsf{cur}}^{\mathsf{cold}},\Delta_{\mathsf{log}}^{\mathsf{cold}}\} + \sum_{\tau_{i,k}\in\tau_{\mathsf{new}}^{2}} \delta_{i,k}^{\mathsf{replay}}\right\} + \sum_{\tau_{i,k}\in\tau_{\tau_{\mathsf{new}}}} \delta^{\mathsf{spawn}}$$

Proof Sketch: First, note that the overhead for n to replay the logs of all new tasks in τ_{new}^{2} is bounded by $\sum_{\tau_{i,k}\in\tau_{\mathsf{new}}^{2}} \delta_{i,k}^{\mathsf{replay}}$. Further, since the transmission of the states (or logs and checkpoints) for the new tasks can take place concurrently with the execution of mode transition actions for existing tasks on n, the maximum delay for n to a) complete the mode transition actions for its existing tasks, b) obtain the latest states of all new tasks in τ_{new}^{1}, and c) obtain the logs and checkpoints of all new tasks in τ_{new}^{2} and then replay the logs is bounded by

$$\max\{\max\{\Delta_{\mathsf{cur}}^{\mathsf{cold}},\, \Delta_{\mathsf{state}}^{\mathsf{cold}}\}, \max\{\Delta_{\mathsf{cur}}^{\mathsf{cold}},\, \Delta_{\mathsf{log}}^{\mathsf{cold}}\} + \sum_{\tau_{i,k}\in\tau_{\mathsf{new}}^{2}} \delta_{i,k}^{\mathsf{replay}}\},$$

which is $\max\left\{\Delta_{\mathsf{state}}^{\mathsf{cold}},\, \max\{\Delta_{\mathsf{cur}}^{\mathsf{cold}},\Delta_{\mathsf{log}}^{\mathsf{cold}}\} + \sum_{\tau_{i,k}\in\tau_{\mathsf{new}}^{2}} \delta_{i,k}^{\mathsf{replay}}\right\}$. By adding this overhead with the overhead to spawn all new tasks on n, which is bounded by $\sum_{\tau_{i,k}\in\tau_{\tau_{\mathsf{new}}}} \delta^{\mathsf{spawn}}$, we obtain the maximum transition overhead of n. Hence, the theorem. ■

VII. The Cascade algorithm

The algorithm we have described so far works fine, but it is not very scalable. In particular, the ILP from Section V-B can become very complex as the number of nodes grows. In this section, we describe an algorithm that is much more scalable.

A. Overview

One simple way to reduce the complexity of the ILP is to break the system into multiple smaller partitions, to allocate each application to a particular partition, and to solve a separate ILP for each partition. This helps because the complexity of the ILP grows superlinearly with the number of nodes and applications; thus, solving a single giant ILP will take far more time than solving k smaller ILPs that each contain roughly $1/k$ of the nodes and applications.

However, there are a few complications. The first has to do with the way applications are allocated to the partitions. At first glance, this looks like an instance of bin packing: we can simply make the allocations based on the computation and bandwidth resources that are available in each partition. However, it is possible that the set of applications is schedulable on the system as a whole but cannot be neatly subdivided into pieces that fit into the partitions – for instance, two partitions might each end up with half an application's worth of slack. In this case, we can simply omit the leftover applications initially, when solving the ILPs for the partitions, and then propagate the solutions to the larger ILP, which thus becomes much simpler (with only a few leftover applications remaining to be scheduled). Occasionally, it may be impossible to fit in the leftover application even using the larger ILP, in which case it has to be dropped; however, since we do the bin packing in criticality order (using first-fit), this problem will only ever affect the lowest-criticality applications.

The second complication is that, if the system is large, the above trick will still leave a fairly complex ILP. We can avoid this problem by doing the partitioning recursively, starting with just a few large partitions (or perhaps even just two) and then repeatedly subdividing each of them until they become small enough to solve. If the solutions are then propagated back up the hierarchy, the ILPs at each level remain small and can all be solved efficiently.

A final complication has to do with the network topology. It is not uncommon for CPS to have a few bottlenecks – say, low-capacity links that bridge between higher-capacity buses. If we are not careful when picking the partitions, we may very well end up with partitions that contain nodes on both sides of such a bottleneck. This would cause a lot of traffic to be routed through the bottleneck, and thus cause the partition-level ILPs to become unsolvable much more quickly than the giant system-level ILP. Thus, we must take care to partition the system such that the partitions are well-connected internally but have as few connections as possible between them.

In summary, the algorithm proceeds roughly as follows:

1) Divide a graph into subgraphs such that the subgraphs are as little connected as possible.

Algorithm 2 Partitioning Algorithm (based on [16])

```
 1: $NoImproveCount = 0$
 2: while $NoImproveCount < totalPasses$ do
 3:     $g_1, g_2 \leftarrow$ random initial partition of G
 4:     $L \leftarrow \{\}$                                  ▷ list of cells moved in this pass
 5:     while $g1, g_2$ both contain at least $Min_{Nodes}$ nodes do
 6:         for $i = 1 \cdots n$ do                        ▷ compute initial gains
 7:             $F(i) \leftarrow$ # of nodes in same partition as $i$
 8:             $T(i) \leftarrow$ # of nodes in different partition as $i$
 9:             $gain(i) \leftarrow F(i) - T(i)$
10:         "Free cell" $C \leftarrow \arg\max_i gain(i)$              ▷ $C \in g_j$
11:         Move $C$ from $g_j$ to $g_{1-j}$.
12:         Add $C$ to $L$.
13:         for $i = 1 \cdots n, i \notin L$ do                       ▷ update gains
14:             $F(i) \leftarrow$ # of nodes in same partition as $i$
15:             $T(i) \leftarrow$ # of nodes in different partition as $i$
16:             $gain(i) \leftarrow F(i) - T(i)$
17:     if $V < V_{MIN}$ and constraints satisfied then
18:         $V_{MIN} = V$
19:         $P = \{g_1, g_2\}$
20:         $NoImproveCount = 0$
21:     else
22:         $NoImproveCount++$
23: return $P$
```

2) Allocate *entire applications* to the bins that each subgraph represents.
3) Subdivide the graphs recursively until some threshold number of nodes remains in each graph.
4) Solve the ILP problem for the leaf-layer subgraphs (in parallel).
5) If, at any point, we find that a schedule is infeasible, we can try to reduce the load by reducing the number of applications we are trying to schedule or by reducing the number of replicas we have.
6) Recursively propagate the computed schedule and any unschedulable applications one layer up and solve the ILP problem for the parent subgraph to find a schedule for applications rejected (if any) at lower-layer subgraphs.

We make two additional observations. First, this approach is not likely to find the optimal solution: the schedule we can generate by combining the various ILP solutions will be slightly worse than the schedule the original giant ILP would have generated. However, as we will show in Section IX, the actual loss tends to be relatively small. Second, although we use partitioning here to efficiently solve the scheduling problem for a *single* mode, it could be useful to use the same partitioning for *all* the modes; this might make it possible to respond to most faults locally within a partition, without having to reconsider the schedules for the other partitions.

B. Graph partitioning

Graph partitioning is itself a hard problem. We can choose to partition the graph globally or locally, but neither is guaranteed to get the optimal solution unless we apply a brute-force approach. One localized method that can run on hypergraphs is the FM algorithm [16] (Algorithm 2), which is an extension of the K-L algorithm [27]. The FM algorithm can handle unbalanced partitions and hyperconnected edges, and it runs in linear time, which makes it a good candidate for our purposes. It also supports a notion of constraints, which we can use, e.g., to enforce that each partition must have some minimum number of nodes.

The algorithm starts by using a graph that has been randomly partitioned. Let us call these two partitions the left and right partitions. Each vertex (node) in the graph is a part of one or more *nets* (links); we can use this feature to represent nodes that have sensors and actuators that are needed for particular applications and thus should be put into the same partition with them. For each node n, we calculate $FS(n)$ and $TE(n)$, where $FS(n)$ is the number of nets for which n is the only node in the left partition and $TE(n)$ is the number of nets containing n that are located entirely in the left partition.

The *gain* of moving node n to the other partition is defined as $FS(n) - TE(n)$. The algorithm takes the node with the highest gain and moves it to the other partition, as long as no constraints are violated by doing so. Once a node is moved, it is fixed to be in the new partition (hence the linear runtime). The gains of the neighbors of the moved node are updated to reflect the new state of the partitions.

To prevent the partitioning algorithm from producing unreasonably small partitions, or partitions that are disconnected internally, we provide a few additional constraints to the partitioning algorithm:

1) Each partition P must have at least $f+1$ nodes
2) There exists a path from any node in a partition P to any other node in that partition
3) No faulty node is a member node of a partition

C. Bin packing

As a first approximation, the bin-packing part of the algorithm is simple enough: we process the applications in decreasing order of criticality and use first-fit to allocate each to a partition. (Notice that we are packing applications into partitions, not into individual nodes; the actual assignments will be done by the ILP.) The condition that determines whether a given candidate application "fits" into a particular partition is that the partition must have sufficient resources available to accommodate all tasks of that application. We sum the density of the task we want to pack in addition to the number of replicas we would like to schedule for it (which is itself a function of the criticality of the application), and compare it to the CPU slack available in the graph.

$$\mathsf{util}(A_i) \times (F_i(f_m)+1) \leq \mathsf{slack}(\mathcal{G}).$$

Here, $\mathsf{util}(A_i)$ denotes the total density of the tasks in the application A_i (the application to be packed) and $\mathsf{slack}(\mathcal{G})$ denotes the total remaining available CPU bandwidth of the nodes in the current subgraph $\mathcal{G}$ after having scheduled applications mapped onto it by child subgraphs.

VIII. Implementation

For our experiments, we implemented a prototype of Cascade in 6,425 lines of C++ code. The implementation includes the ILP described in Section V, as well as the hierarchical bin-packing described in Section VII. Since the hierarchical ILP uses smaller versions of the original, giant ILP, we can easily compare Cascade to a strawman approach that solves the giant ILP directly; we will refer to this as "pure ILP". To solve the ILPs, we invoke Gurobi [22], a well-known commercial solver. We use a timeout of one hour for each individual ILP.

For comparison, we also implemented TPCD [3], an efficient heuristic that can pack a set of replicated tasks onto a set of nodes. A key difference between Cascade and TPCD, as well as its variants TPCDC [3] and TPCDC+R [4], is that the latter are static – that is, they do not change their response based on what faults have occurred – and that they assume a fully-connected network with no bandwidth limitations. As such, the scheduling problem for TPCD and its variants is somewhat easier, but still close enough to serve as a basis for comparisons.

Briefly, the input to TPCD is a set of independent tasks. Each task has a primary and a configurable number of replicas; in our implementation, we use 3 replicas for criticality-0 tasks, 2 replicas for criticality-1, and 1 replica for criticality-2. All replicas are "hot" replicas, so their utilization is identical to that of their primary. TPCD uses a best-fit decreasing packing algorithm, with the constraint that a task's replica cannot be on the same node as its primary or another replica of the same task. It packs tasks in decreasing order of replication. That is, tasks with three replicas will have their third replicas packed prior to all second replicas. Primaries are packed last. If TPCD runs out of space and cannot find a node with enough spare utilization for a given replica, it will create a new node. Since this is not an option in our setting, where the number of nodes is fixed, we instead drop any tasks that do not "fit" into the system once it has filled up.

IX. Evaluation

Our evaluation was designed to answer three high-level questions: 1) Can we leverage Cascade to find near optimal solution that are scalable? 2) How well can such a task allocation strategy handle the introduction of faults? And 3) How well does Cascade compete with existing work?

All our experiments ran on Intel Xeon E5-2620 v3 servers, each with 24 hyperthreaded cores (two hyperthreads per physical core) with 64 GB RAM. Each machine ran Fedora 26.

To compare the pure ILP with Cascade, we generated systems with networks that were either fully connected (for Section IX-A) or generated using the Erdös-Renyi G(n,p) model, with p=0.5 – that is, each node-to-node link had a 50% chance of being included in the system. We used 100 random seeds to vary the input graph and input application set that we wanted to schedule. Applications consisted of single tasks (i.e., no dependencies); we drew both their CPU utilizations (between 0.1 and 0.7) and their criticality level (0, 1, or 2) uniformly at random, and we stopped generating applications as soon as adding the next one would have caused the total utilization to reach or exceed the number of nodes N. We selected the period uniformly at random from the range [10,

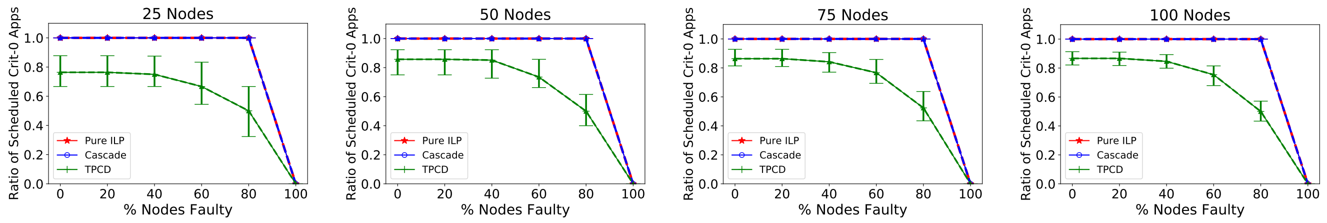

Fig. 2: Schedulability for criticality-0 applications using Cascade, TPCD, and our strawman approach (Pure ILP). The points show the median, and the error bars show the 20th and the 80th percentile.

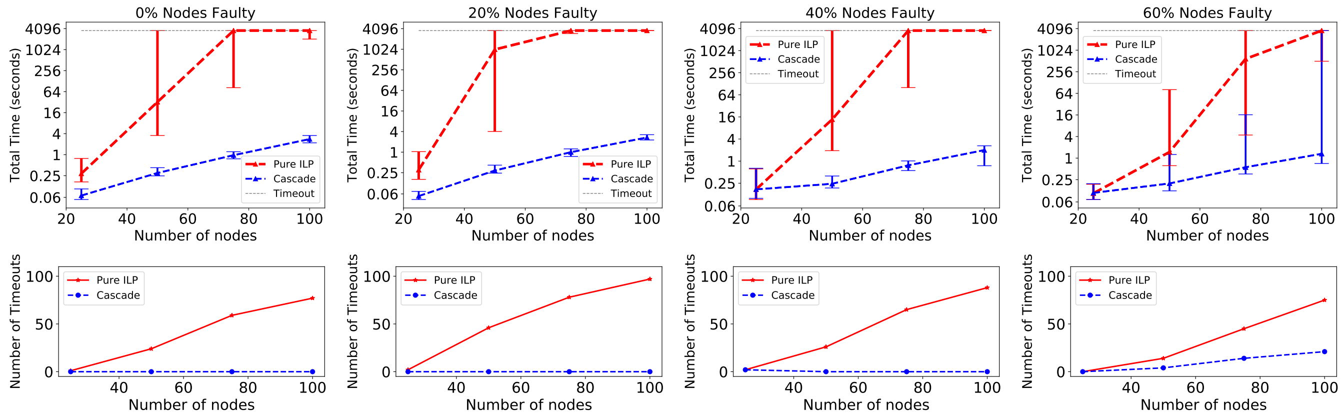

Fig. 3: Top row: Time needed to allocate tasks using Pure ILP and Cascade, for systems of different sizes and with different fractions of faulty nodes. Bottom row: Timeout rates for both techniques.

40] ms, and we set the deadline equal to the period. We set $R_{\max} = 100$ms and $P_{\mathrm{ft}} = 5$ms. The base of the exponentiation in the objective function was set to $M = 1.2$. The execution time was calculated as the selected CPU utilization multiplied by the period. Criticality-0 applications were required to start with three replicas (that is, four copies of each task), and could thus tolerate up to three simultaneous faults. Criticality-1 applications started with two replicas, and criticality-2 applications with a single replica.

A. *Comparison to TPCD*

Our first experiment is designed to get a general impression of the quality of the schedules Cascade produces. We created systems of sizes $25, 50, 75$, and 100 nodes and a fully-connected network (to enable comparisons to TPCD, which requires this) and then used Pure ILP, Cascade, and TPCD to generate schedules for random modes in which between 0 and 60% of nodes have failed. We discarded networks where the solver timed out. Figure 2 shows our results (median across all seeds). Cascade ensures that *all* the criticality-0 applications are scheduled until almost all of the nodes in the system are faulty, whereas TPCD cannot schedule them all even in the absence of faults. To be fair, TPCD was designed with the ability to add nodes to the system when more are needed, whereas Cascade assumes that the number of nodes is fixed and that it must find the best possible schedule using the available resources.

B. *Scheduling time*

Next, we examined how much Cascade improves the scheduling time relative to using a single giant ILP. We generated systems of different sizes, ranging from 25..100 nodes. We solved each system with 0%, 20%, 40%, and 60% faulty nodes, using both the gigantic ("pure") ILP and Cascade. We did not use TPCD for this experiment because it requires a fully-connected network.

Figure 3 shows the results; notice that the vertical axis is logarithmic. As expected, the runtime for the pure ILP increases very quickly with the size of the system; as early as with N=50 nodes, the 80^{th} percentile is at the one-hour time limit we imposed. In contrast, the runtime for Cascade grows slowly with the system size, as long as the fraction of faulty nodes is not too large. For 60% faulty nodes and above, the graph often is no longer partitionable with our method, so Cascade's performance starts to approach that of Pure ILP.

Due to large runtimes, we did not attempt to run the pure ILP for systems larger than N=100 nodes, but we ran additional experiments with larger systems using Cascade. Fig. 4 shows the results, again with a logarithmic vertical axis. The difficulty of the scheduling problem can vary substantially across our randomly chosen test graphs; because of this, the 80th percentile is much higher than the median (but timeouts occurred for only four of the topologies). Overall, the solution time does increase with the system size, but the numbers are orders of magnitude smaller than those for Pure ILP.

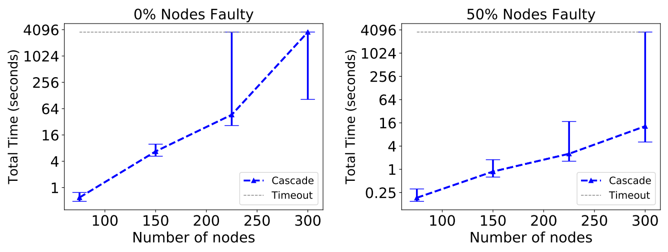

Fig. 4: Time needed to solve Cascade's hierarchical ILP for larger systems. The points show the median, and the error bars show the 20th and the 80th percentile.

Although we were unable to compare Cascade to TPCD (because our network topologies were not fully connected), we wanted to include at least a qualitative comparison, so we ran experiments on some fully-connected networks. On these networks, TPCD is very fast: it can finish a task allocation for a 1,000-node system in about four seconds.

C. Schedule quality

Another possible concern is that, while Cascade can find a solution much faster than the "pure" ILP, the quality of the resulting schedules might suffer. To examine this, we performed the following experiment. We generated 100 random graphs with $N = 25..100$ nodes, and we scheduled them with $0\%, 20\%, 40\%$, and 60% faulty nodes, using both Pure ILP and Cascade. For the graphs that did not time out, we then measured the average fraction of the tasks at each criticality level that were schedulable, as well as the average value of the objective function for the ILPs that we were able to solve within our one-hour cut-off.

Figure 5 shows, for $N = 25, 50, 75, 100$, the median schedulability at each criticality level. We make three key observations. First, the most critical tasks (at criticality level zero) remain completely schedulable with both approaches, even when 60% of nodes are faulty. Second, most of the lower-criticality tasks remain schedulable as well, with the criticality-2 tasks suffering first. This shows Cascade's ability to dynamically reduce the number of replicas for a given application once there are no longer enough nodes to support the original replication level. Finally, Cascade does do a little worse than the pure ILP, but only at the lower criticality levels, and not by much.

Figure 6 shows a different view on this data: it compares the objective function for both Pure ILP and Cascade at the 10^{th} to 90^{th} percentiles in 10 percentile increments. Overall, the numbers are roughly comparable across all system sizes and failure scenarios.

D. Worst-case failure scenario

So far, we have compared Cascade and TPCD only in terms of runtime and schedulability. This does not capture an important advantage of Cascade: whereas TPCD statically allocates tasks to nodes, Cascade can dynamically adapt to different failure scenarios.

TABLE I: Assignments generated for a small graph by TPCD. "P" indicates primary and "R" indicates replica.

		Node ID					
		0	1	2	3	4	5
App ID	0	R	-	R	-	-	P
	1	-	R	-	-	P	-
	2	-	-	-	R	-	P
	3	R	-	R	-	P	-
	4	-	-	-	-	-	-
	5	-	R	-	R	-	P

		Sets of Faulty Nodes														
		{5}			{3,5}			{1,3,5}			{1,2,3,5}			{0,1,2,3,5}		
		P	R_1	R_2	P	R_1	R_2	P	R_1	R_2	P	R_1	R_2	P	R_1	R_2
TPCD	Crit. 0	3	3	1	3	2	1	2	2	1	2	1	0	1	0	0
	Crit. 1	2	1	0	1	1	0	1	0	0	1	0	0	1	0	0
Cascade	Crit. 0	3	3	2	3	3	0	3	3	0	3	0	0	2	0	0
	Crit. 1	1	1	0	3	0	0	1	0	0	1	0	0	1	0	0

TABLE II: Number of applications that can retain their primary (P), first replica (R_1), and second replica (R_2) in an example network with six nodes and three applications.

To illustrate this point, we used a single, very small system with only six nodes and a fully connected network. Table I shows the task allocation that TPCD generates for this system; applications 0, 3, and 5 have a primary (P) and two replicas (R) because their criticality is higher; the other applications only have a primary and a single replica.

Now consider the scenario where node 5 fails, then node 3, and finally node 1. In this situation, application 5 no longer works, since its primary and both its replicas were mapped to the nodes that have failed. Since the faults did not occur simultaneously, there would have been time after the first fault to reconfigure the system and create another replica on a different node. TPCD uses a static mapping of tasks to nodes and thus cannot support such a reconfiguration, but Cascade does take advantage of this opportunity, and can thus survive the second fault without losing the high-criticality application.

We extended the comparison until Cascade was unable to schedule a criticality-0 application, and we show the results in Table II; we report, for each failure scenario, the number of high-criticality and low-criticality tasks and replicas that are still active. As in the previous example, Cascade prioritizes the high-criticality applications and spawns additional replicas for them (potentially at the expense of lower-criticality applications) as soon as a fault is detected, whereas TPCD, with its static task mapping, is not able to respond to faults. As a result, in 4/5 scenarios, Cascade manages to continue to run all of the high-criticality applications; in the fifth scenario, the system loses five of the six nodes (83%) and is no longer able to support them all. In contrast, TPCD's success rates vary, but the inability to adapt prevents high-criticality applications from running even in cases where there are enough resources.

E. Case study: Volvo XC90

Finally, we wanted to see how well Cascade would do on a topology from a real cyber-physical system. We chose the onboard network of the Volvo XC90 for this case study;

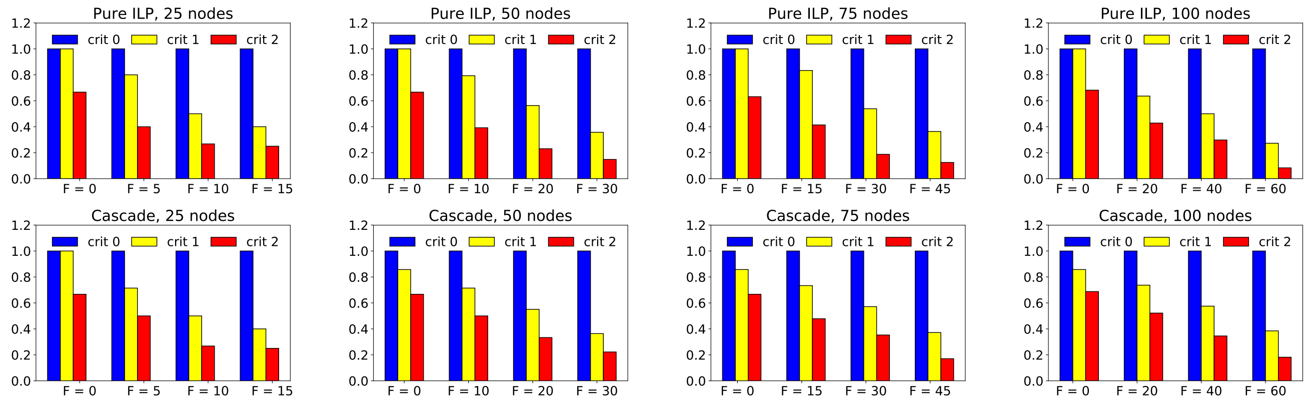
Fig. 5: Fraction of applications that can be scheduled, for Erdös-Renyi networks with 25..100 nodes, using both pure ILP and Cascade, as the number of faults increases. Timeouts have been filtered out for both techniques.

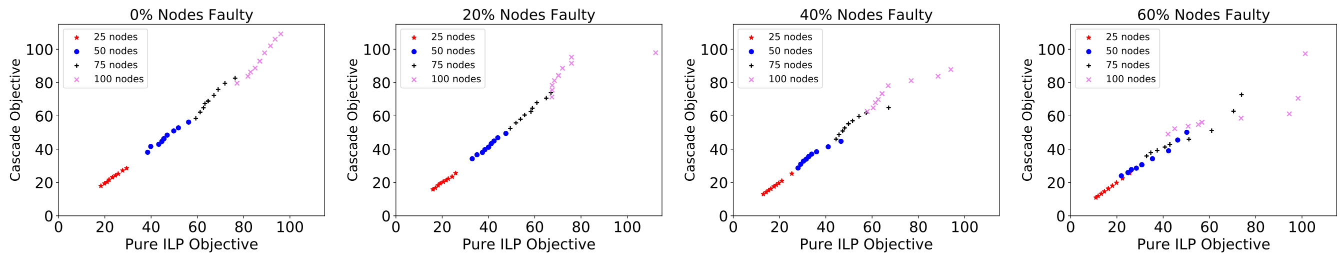
Fig. 6: Objective function comparison of pure ILP (red) and Cascade (blue) for Erdös-Renyi networks of size 25..100 as the percentage of nodes in the system that are faulty increase. Points are selected as $10^{th}..90^{th}$ percentiles.

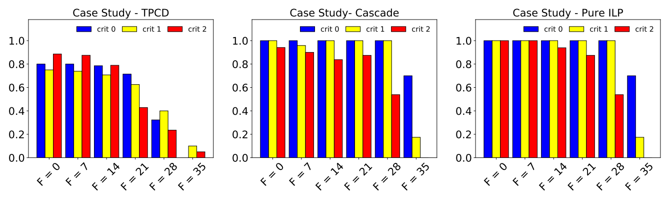
Fig. 7: Fraction of applications scheduled for the Volvo XC90 network, using TPCD (left), Cascade (middle), and Pure ILP (right), for different numbers of faults.

we have already shown its topology earlier, in Figure 1. This network contains 38 compute nodes and 13 buses – 1 HCAN, 1 LCAN, 1 MOST, and 10 LIN – for connectivity. We used the same workload generation method as for the other experiments, with two differences: first, to get at least a rough sense of how well TPCD would do, we first generated TPCD application sets for each of the buses (which can be thought as the fully connected network TPCD requires), and we then merged these application sets to form the workload for Cascade. And second, since some of the buses are very small and cannot accommodate a criticality-0 application with three replicas, we reduced the number of replicas by one at each level – that is, two replicas for criticality-0, one for criticality-1, and zero (i.e., no redundancy) for criticality-2. As before, we measured how the number of scheduled applications changed as the number of faults increases.

Figure 7 shows our results. The behavior is consistent with the earlier results: since TPCD uses a static task mapping and cannot respond to faults by reconfiguring the system, it starts to lose criticality-0 applications as soon as a fault knocks out all of their original replicas. Cascade loses the least critical applications more quickly, since it uses their resources to replenish the replica sets of the more critical applications, but, in return, the most critical applications survive far longer – they start failing only after almost all of the nodes are gone.

X. Conclusion

Bounded-time recovery is a new addition to the community's fault-tolerance toolbox; it has a number of interesting properties, but also comes with some new challenges. In this paper, we have focused on one of them: its very unusual scheduling problem, which is quite different from anything else we had seen before. Although the problem appears very complex at first, Cascade shows that this complexity can be managed quite well, and that it is possible to efficiently find schedules for fairly large systems, with more than 100 nodes.

Cascade is probably not the last word on BTR scheduling; for instance, by using the mode transitions "only" to move replicas or to create new ones (and not, say, to change what the system is doing, or how it is doing it), it leaves a lot of potential flexibility on the table. Nevertheless, Cascade should provide a solid foundation for more powerful BTR scheduling techniques.

Acknowledgements

We thank Alice Cheng for her help with an initial implementation of the Cascade algorithm and the anonymous reviewers for their thoughtful comments and suggestions. This research was supported in part by NSF CNS 1750158, CNS 1703936 and CNS 1563873, by ONR N00014-20-1-2744, and by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

References

[1] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–3–4, 1993.

[2] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, D. C. Schmidt, C. Lu, and C. Gill. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *Proc. RTAS*, 2010.

[3] A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *Proc. RTAS*, 2017.

[4] A. Bhat, S. Samii, and R. Rajkumar. Recovery time considerations in real-time systems employing software fault tolerance. In *Proc. ECRTS*, 2018.

[5] G. M. Brown, M. G. Gouda, and C.-L. Wu. Token systems that self-stabilize. *IEEE Trans. Comput.*, 38(6):845–852, June 1989.

[6] J. E. Burns and J. K. Pachl. Uniform self-stabilizing rings. *ACM Transations on Programming Languages and Systems*, 11(2):330–344, Apr. 1989.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[9] A. Chen, H. Xiao, L. T. X. Phan, and A. Haeberlen. Fault tolerance and the five-second rule. In *Proc. HotOS*, May 2015.

[10] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proc. USENIX ATC*, 2012.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. NSDI*, 2008.

[12] M. Di Natale and J. Stankovic. Dynamic end-to-end guarantees in distributed real time systems. In *Proc. RTSS*, 1994.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov. 1974.

[14] S. Dolev. *Self-Stabiliaztion*. MIT Press, 2000.

[15] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. PODC*, 1990.

[16] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. DAC*, 1982.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[18] G. Fohler. Changing operational modes in the context of pre-runtime scheduling. *IEICE Transactions on Information and Systems*, E76-D(11):1333–1340, 1993.

[19] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE TPDS*, 8(3), Mar. 1997.

[20] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, Apr. 1991.

[21] M. G. Gouda, R. R. Howell, and L. E. Rosier. The instability of self-stabilization. *Acta Informatica*, 27(8):697–724, 1990.

[22] Gurobi Optimization, Inc. http://www.gurobi.com.

[23] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proc. OPODIS*, 2009.

[24] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. PODC*, 1990.

[25] H. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *Proc. ICDCS*, 1993.

[26] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, Nov. 1993.

[27] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[28] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[29] H. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, 1979.

[30] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[31] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[32] D. Marinca, P. Minet, and L. George. Analysis of deadline assignment methods in distributed real-time systems. *Computer Communications*, 27(15):1412–1423, 2004.

[33] M. Morari. Fast model predictive control (MPC). Presentation, available from http://divf.eng.cam.ac.uk/cfes/pub/Main/Presentations/Morari.pdf.

[34] T. Nolte. *Share-driven scheduling of embedded networks*. PhD thesis, Mälardalen University, 2006.

[35] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX ATC*,

2014.

[36] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proc. ECRTS*, 1998.

[37] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[38] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[39] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or EDF scheduling. In *Proc. DATE*, 2009.

[40] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority preemptively scheduled systems. In *Proc. RTSS*, 1992.

[41] Y. Zhang and J. Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual reviews in control*, (32):229–252, 2008.