Right On Time Distributed Shared Memory

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

David Kozhaya
EPFL, Switzerland
david.kozhaya@epfl.ch

Yvonne-Anne Pignolet
ABB Corporate Research, Switzerland
yvonne-anne.pignolet@ch.abb.com

***Abstract*—The demand for real-time data storage in distributed control systems (DCSs) is growing. Yet, providing real-time DCS guarantees is challenging, especially when more and more sensor and actuator devices are connected to industrial plants and message loss needs to be taken into account.**

In this paper, we investigate how to build a shared memory abstraction for DCSs as a first step towards implementing different shared storage systems in a DCS context. We first prove that, in the presence of host crashes and message losses, the necessary guarantees of such an abstraction are impossible to implement using a traditional approach that has no access to the internals of existing DCS services, e.g., a modular approach where algorithms are built on top of existing software blocks like failure detectors. We propose a white-box approach that utilizes messages of existing services in any DCS as the sole means of communication. More precisely, we present *TapeWorm*, an algorithm that attaches itself to the heartbeat messages of the failure detector component in DCSs. We prove that *TapeWorm* implements the desired shared memory guarantees for applications running on a DCS. We also analyze the performance of *TapeWorm* and we showcase ways of adapting *TapeWorm* to various application needs and workloads.

***Keywords*-real-time distributed shared memory; probabilistic losses; distributed control systems; failure detection;**

I. Introduction

In multi-core machines, shared memory (at the hardware level) constitutes the typical means of communication for hosts (processes) [1]–[3]. In message-passing distributed systems, however, hosts communicate by exchanging messages over a network. Hence, shared memory, in such distributed systems, no longer physically exists but rather becomes a distributed communication abstraction built using message exchange and local memories of hosts [1]–[5].

A distributed shared memory abstraction constitutes a basic building block for implementing networked storage systems, distributed file systems and distributed key-value stores. These distributed data services are undeniably demanded in distributed control systems (DCSs) [6]–[11]. For example, power grid DCSs require real-time distributed storage systems (distributed hash tables) to store and retrieve monitoring data for wind and photo-voltaic generation sources [6]; ship-board DCSs require distributed real-time data services to be embedded within the ship [7], [8]; traffic control and agile manufacturing require the presence of distributed fresh data that reflects real-world status [7]–[10]; real-time execution platforms for DCSs, such as [11], rely on real-time replicated data structures for maintaining system and crash detection information.

In addition to its importance as a building block for various data storage services, shared memory is of immense benefit to application programmers in DCSs; programming with shared memory is considered significantly easier than working with message exchanges [2]. This programming simplicity encourages having more control application programmers and limits programming errors. Also, algorithms designed for shared memory in mind could thus be directly used in a message-passing context that provides the distributed shared memory abstraction. In short, there is a fundamental need to study real-time data abstractions, such as shared memory, in DCSs.

To this end, we investigate in this paper how to build a shared memory abstraction for DCSs. We first derive the guarantees that need to be provided by *read* and *write* operations accessing shared memory in a DCS context [1]–[5]; such guarantees define the respective consistency level. We show (Section II-D) that the needed requirements necessitate the presence of all of the following properties: *real-time termination*, *agreement* and *freshness*. Roughly speaking, real-time termination means that each operation, be it a read or a write, always completes in a bounded known duration. Agreement ensures that read operations, issued within a fixed known time-window to the same shared object, return the same value. Freshness guarantees that any value returned by a read operation is a value written by one of the last completed "c" writes, where c is a fixed known number.

The necessary consistency level, captured by the three aforementioned properties, is, however, challenging to implement when accounting for host crashes and message losses that can happen in a DCS. Control systems typically experience host crash rates of about $10^{-5}/hr$ and link failure rates (causing message loss) in the range of $10^{-5}/hr$ (permanent failures) and $10^{-3}/hr$ (transient failures) [12], [13]. In fact, we prove that the three properties listed above are impossible to achieve using traditional ways for implementing distributed shared storage [4], [8]–[10], i.e., using algorithms that do not have access to the internals of DCS services, for example, a black-box approach where algorithms are built on top of existing failure detector software blocks (more details in Section III). Yet data storage services for DCSs are imminently needed [6]–[10].

To circumvent this impossibility, we propose a white box approach that utilizes existing services running within DCSs [11], [14], namely failure detection. DCSs usually employ failure detectors that provide means for detecting host crashes in real time [14]–[18]. More specifically, failure detectors output a list of hosts suspected to have crashed. Accordingly, DCSs employ recovery mechanisms requiring to shift application-related tasks to be executed only by those hosts which are not suspected to have crashed. In this sense, the output of suspected hosts (if any exists), even if these hosts have not actually crashed, is no longer visible to applications [14]. Benefiting from this behavior, we propose a solution that guarantees the DCS consistency level (consisting of the three aforementioned properties) as seen by the applications, and not necessarily within the set of all non-crashed hosts.

We design *TapeWorm*, an algorithm that attaches itself to the crash monitoring messages (heartbeats) typically exchanged on a periodical basis to detect host crashes in DCSs [11], [12], [14]. *TapeWorm* uses the underlying heartbeats as the sole means of transporting information. *TapeWorm*, thus, benefits from the real-time operation of failure detectors in DCSs in the following sense. Crashed hosts, based on the exchanged heartbeats, are always suspected in real time (a deterministic guarantee) [14]–[18]. The speed of detecting crashed hosts (detection time) should be fast enough to guarantee that applications can recover from potential host crashes and still meet their deadlines. *TapeWorm*, by using heartbeats as a transportation mechanism, can have the leverage of reaching hosts that are not suspected, providing services to such hosts in real-time.

We prove that *Tapeworm* indeed implements the required three properties of shared memory among non-suspected hosts. We show that *TapeWorm* can be adapted, if need be, to provide real-time guarantees faster than those of the failure detector it relies on. Precisely, *TapeWorm* can be adapted to allow read operations to return fresher values in the allowable freshness range[1]. We also conduct a mathematical analysis computing the probability distribution on the freshness of the values returned by *TapeWorm* as well as the respective incurred bandwidth cost. Our analysis quantifies *TapeWorm*'s performance in terms of variable system parameters, such as the size of the system and the message loss rate. We also devise an optimization of *TapeWorm* for static workloads, where the time at which operations are invoked is known by all hosts in the system.

In summary, the main contributions of this paper are:

1) A first precise derivation of the necessary guarantees that a shared memory abstraction must provide in DCSs.
2) Theoretical proofs showing the impossibility of implementing such guarantees using traditional approaches [4], [5], [8]–[10], e.g., using a black-box approach.
3) *TapeWorm*, an algorithm that circumvents the above impossibility by following a white-box approach directly utilizing failure detector algorithms of DCSs. *TapeWorm* implements the required shared memory guarantees for applications running in a DCS.
4) A mathematical analysis quantifying the performance of *TapeWorm* and showcasing ways of adapting and optimizing *TapeWorm* respectively to application needs and workloads.

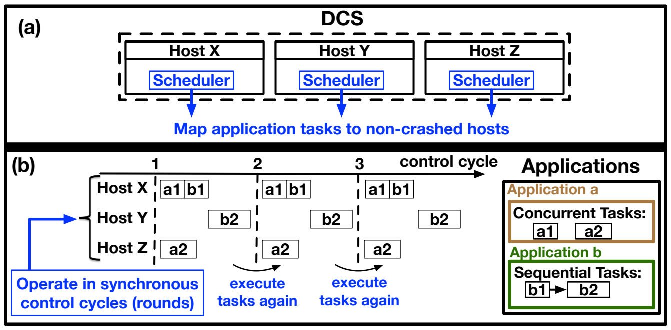

Figure 1. A DCS with three hosts running two control applications.

The rest of the paper is organized as follows. Section II details how a DCS operates and identifies the required properties for a DCS shared memory abstraction. Section III highlights the difficulty of implementing such properties by proving two impossibility results. Section IV presents our algorithm, *TapeWorm*, for implementing shared memory in a DCS, proves *TapeWorm*'s correctness and shows how to return fresher values using *TapeWorm*. Section V presents an analysis of *TapeWorm*'s performance, precisely analyzing the freshness of the values returned as well as the incurred bandwidth. Section V demonstrates as well an optimization of *TapeWorm* under static workloads. Section VI discusses related work, while Section VII concludes the paper.

II. DCSs For Cyclic Control Applications

A distributed control system (DCS) consists of a set of n hosts (or processing units), denoted by $\Pi = \{h_1, h_2, ..., h_n\}$. As in any distributed system, these hosts can *fail (crash)* [2], i.e., stop executing operations. The rate of host failures in control systems is typically around $10^{-5}/hr$ [12], [13]. Hosts are considered to be synchronous, i.e., the delay d_p of performing a local step has a fixed known bound. Hosts have access to local synchronized clocks with bounded skew. Using these local clocks, hosts define control cycles (rounds) of the same fixed duration. The cycle duration is $>> d_p$. These control cycles are time-wise synchronized among all hosts, i.e., the start and end of a cycle occur at all hosts at the same time (with a bounded skew). Cycle lengths vary according to applications, e.g., 8-10 ms for substation automation and low-level robot interfaces and up to 1 s for temperature-driven applications [19].

DCSs often execute control applications that are cyclic [11], [20], [21], i.e., run periodically. For example, an application might be required to periodically read values from a sensor and to activate "intensive cooling mechanisms" after t seconds of a machine exceeding a

[1] The freshness of writes is guaranteed since writes are typically issued for example by sensors on a periodical basis.

certain temperature threshold. Specifically, cyclic control applications consist of several small *tasks* that repeatedly execute. Some of these tasks run concurrently on several hosts, possibly on behalf of different control applications (see Figure 1). In every cycle, each host executes the tasks assigned to it by the *scheduler*.

A. Scheduler

The scheduler is a distributed system module that specifies which application tasks run on which hosts and in what order (see Figure 1). The allocation of tasks to hosts is called a *configuration*. A scheduler makes sure that all hosts can execute the assigned tasks without exceeding the total cycle duration. Moreover, the scheduler ensures that configurations allow all applications to run correctly and to meet their deadlines. Upon detecting the crash of a host, the scheduler computes (taking into account the crashed host) a new configuration, which maps tasks to hosts. This re-mapping of tasks ensures that applications meet their deadlines despite host crashes.

B. Communication

A pair of hosts in a DCS is connected by two logical unidirectional links. Hosts h_i and h_j are connected by links l_{ij} and l_{ji}. Links, in this context, can abstract a physical bus or a dedicated network link. Arguably, all communication is prone to random disturbances, for example, bad channel quality, interference, collisions, stack overflows etc [22]. Messages can thus be lost. When there is no loss, we assume that messages have a bounded delay, say d. We specifically consider that a message sent on link l_{ij}, $\forall i \neq j$, at any time t has probability $0 < P_{ij}(t) < 1$ of getting lost. Configurations computed by the scheduler account for the message delay d. As such, any message scheduled to be sent in a control cycle r, if not lost, is assumed to be received in cycle r.

Sending a message reliably from one host to another, however, can take an unbounded amount of time, due to losses and the required follow-up re-transmissions.

C. Failure Detection and Monitoring in a DCS

A failure detector is a distributed module that runs on every host and provides the DCS scheduler with information about host crashes [23]. The most common monitoring scheme used by failure detectors in DCSs dictates that each host periodically, i.e., in every cycle, broadcasts a heartbeat message of some structure[2] [11], [12], [14], [25]–[27]. Based on these heartbeats and using time-outs, a failure detector monitors which hosts in the system have crashed and which have not. As such, we consider, in this paper, failure detector algorithms using heartbeats and time-outs alone.

Detecting crashes in real time in a DCS is crucial, for instance, in order to allow the scheduler to re-map the tasks (initially assigned to the crashed host) to other hosts, without violating application deadlines. Failure detection varies depending on application needs, but is often expected to be in the order of milliseconds. Since communication is prone to losses, real-time failure detection using heartbeats and time-outs cannot be always *accurate* [15]–[18], [28], [29]. Accuracy depicts a failure detector's ability of not suspecting correct hosts, i.e., hosts that do not crash. As such, at any cycle r, a host can exist in one of the following sets (assuming the fail-stop crash model [23], i.e., no recoveries):

Crashed hosts $\nabla\mathcal{C}(r)$: this set includes all hosts that have crashed at some cycle up to cycle r (included).

Eliminated hosts $\nabla\mathcal{E}(r)$: this set includes all hosts that have not crashed up to and including cycle r but are suspected by the failure detector during cycle r.

Alive hosts $\nabla\mathcal{A}(r)$: is the set of hosts that have not crashed up to and including cycle r and are not suspected during r.

A scheduler considers all hosts belonging to the set $\nabla\mathcal{A}(r)$ functional at cycle r and computes configurations accordingly. All other hosts, i.e., hosts $\in \{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{C}(r)\}$, are considered non-functional, and as such no application tasks are allocated to execute on them [11] or their output (if any exists) is made hidden from the applications [14]. Assuming no recoveries, if $r < r'$, then $\nabla\mathcal{C}(r) \subseteq \nabla\mathcal{C}(r')$.

The scheduler, being a distributed module executing on every host (recall Figure 1), should maintain a consistent state at all cycles. A consistent scheduler state is achieved when the failure detector, at every host, suspects the same set of hosts, and thus provides the scheduler module at every host with the same set $\nabla\mathcal{A}(r)\ \forall\ r$ [11]. An inconsistent scheduler state means that hosts might be executing different configurations (since hosts have different $\nabla\mathcal{A}(r)$). Different configurations mean different mapping of tasks to hosts. In other words, the DCS would be experiencing "downtime" (normal operation is halted) since applications might be executing incorrectly (communication or the order of execution between tasks of the same application might be invalid) [11].

D. Shared Storage Abstraction

Applications in distributed control environments require shared memory functionalities [6]–[11], as application tasks executing on different hosts may require to communicate by reading and writing to shared memory. Shared memory can be viewed as a collection of shared object abstractions. We consider a DCS, as in [11], [30], where a single object is assigned to every task that writes a value. As such, every shared object in a DCS is a read/write object which can be written to by a single host, however, it can be read from by any number of hosts, i.e., single-writer multiple-reader (SWMR) object [1]–[5] (see Figure 2). For simplicity, we assume that a shared object is written to, once every cycle[3].

[2] Different failure detection algorithms may send different information within heartbeats [15]–[18], [24].

[3] From the reader nodes' perspective, multiple writes to an object in cycle r, can be viewed as a single write, that being the last write in r which is available for reads in cycle $r + 1$.

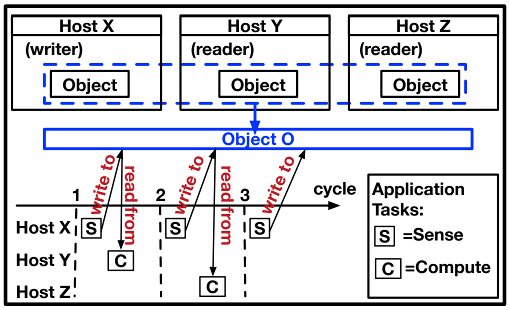

Figure 2. An example of a shared memory (an object) in a DCS.

1) Properties of a DCS Shared Memory. We now determine the properties that should be satisfied by the read and write operations issued to shared memory.

Termination. A DCS requires operations (read and write), to complete within a bounded amount of time after being invoked, say t_{op}. When computing configurations, the scheduler accounts for t_{op} and assigns tasks to hosts accordingly. Cycle durations are defined such that operations, with a delay of t_{op}, complete and return in the same cycle in which they were invoked. Having operations with unbounded delay makes the scheduler's job, of computing configurations with a fixed cycle duration, impossible.

Read Agreement. Certain critical control applications, e.g., those for power system control, require specific consistency on the values being read by hosts in the same cycle. Namely, read operations invoked in the same cycle to the same shared object, upon completion, are required to return the same value. To better illustrate the need for such a requirement, consider a control application where hosts are required to open or close circuit breakers based on the values read from shared memory. If, in some cycle, two or more hosts read different values of the same shared object, these hosts might allow for undesirable power flows leading to blackouts, overheating wires and creating islands [31].

Read Freshness. In a DCS, hosts requiring to read shared memory can typically tolerate some freshness range for the value being read. In other words, it is acceptable if a read operation does not return the latest value written. Such tolerance is typically supported by a DCS due to the presence of failures, message losses, potential unanticipated system delays, etc. More specifically, a read operation invoked by a host in some cycle, upon completion, is required to return a value that is written at most c cycles ago. In practice, the value of c is correlated with the *reaction time*[4] needed by certain applications. Recall, that we assume that every shared object is written to once every cycle.

To sum-up, the consistency of a DCS shared memory is governed by three main properties, formally stated below:

Termination: Any operation invoked by a non-crashed host completes in the same cycle in which it was invoked. The delay between the time an operation is invoked and the time it completes is at most t_{op}.

Agreement: In a given cycle, read operations to the same shared object return, upon completion, the same value.

Freshness: A read operation invoked during cycle r, upon completion, returns a value that is written at most c cycles ago, if the writer host of that object is not suspected at r. If the writer, however, is suspected at some cycle r', then all reads invoked in cycles $\geq r'$ return a value written at some cycle in $[r' - c, r']$.

It is important to note that it is sufficient that services in a DCS, such as shared memory, ensure their respective properties at times when the scheduler state in the DCS is consistent, i.e., when the scheduler at all non-crashed hosts sees the same $\nabla\mathcal{A}(r)\ \forall\ r$. Ensuring shared memory properties, in the absence of scheduler consistency is not required, as the DCS would be down (unavailable to deliver correct services).

In fact, production DCSs, such as [11], [30], adhere to the architecture and mode of operation that we consider in this paper. In fact our distributed shared memory abstraction is inspired by the constraints and requirements governing such industrial DCSs. Nowadays, application areas of programmable logic controllers (PLCs), DCSs and SCADA overlap and include monitoring and control applications for factory automation, substation automation and smart grids [32].

2) Comparing with Classic Abstractions. We now compare the aforementioned properties with those of classic shared objects and related abstractions [4], [5], [7]–[9], [33].

Atomic Objects. Informally, the atomicity property requires that each operation appears as if it was executed instantaneously at some point in time, regardless of the time taken by each operation to complete [4], [5]. Besides atomicity, atomic objects also require that operations respect their temporal order, basically, having reads return the last written value. The properties of a DCS shared object, however, require operations to complete in real time and do not require reads to return the last value written but rather return a value written within a bounded past duration from the time a read is invoked.

Temporally Consistent Objects. Temporal consistency represents the consistency level adopted in most real-time distributed databases [7]–[9]. The temporal consistency property is typically used to quantify the freshness of replicated values among distributed hosts. Two objects are said to be temporally consistent with each other if their corresponding timestamps are within a known fixed bound δ. In this sense, the freshness property defined here ensures temporal consistency among hosts in a DCS. Besides temporal consistency, real-time databases require real-time responses. Real-time response is similar to the termination property defined in this paper. However, the termination property for DCSs is strictly stronger as it requires operations to complete within a bounded delay at all times

[4] Reaction time is the time interval from the moment some value is written until all hosts can read this value.

rather than with high probability. The main difference is that real-time distributed databases do not require the agreement property defined above, essential in a DCS context.

Real-Time Reliable Broadcast. Since we consider SWMR shared objects, we highlight the difference with closely related abstractions like a reliable broadcast abstraction [33]. Roughly speaking, real-time reliable broadcast requires a sent message to be delivered to all hosts or none in some bounded time δ. One main difference is that the shared memory abstraction we define requires not only to agree about the value to be delivered but on the time of delivery as well. Hosts invoking reads in the same cycle should deliver the same value; otherwise we do not require agreement. This means that messages should be delivered within a certain time bound after being sent, but more importantly messages should be delivered within the same cycle at all hosts (requiring to see that message). Another difference is the fact that a real-time reliable broadcast alone cannot ensure the freshness property; it is possible that every message sent is delivered by no one.

III. Feasibility of Implementing Shared Memory in a DCS

We discuss, in what follows, the feasibility of implementing a shared object, satisfying termination, agreement and freshness, in a DCS. We first introduce the following lemma, which we rely on in our proofs later in this section.

Lemma 1. *Any algorithm that deterministically implements the termination property implements "local" operations; operations invoked by a host h_i do not wait for responses from any other host h_j $(\forall i \neq j)$ in order to complete.*

Proof: By contradiction assume that an algorithm implements termination and when a correct host h_i (i.e., does not crash) invokes an operation, h_i waits for a reply from at least one other host h_j $(i \neq j)$ in the system. Since cycle durations are fixed, a host can hence send a finite number of messages within a cycle, say m messages.

We compute in what follows the probability that host h_i loses all m messages sent to it by any host h_j in the system. Recall that $P_{ji}(t)$ is the probability with which the link l_{ji} loses a message at time t $\forall j \neq i$. Let $P_{ji}(t \cap t')$ be the probability that l_{ji} loses the messages (if any is sent) at time t and time t'. Since $0 < P_{ji}(t) < 1$ $\forall t$, then

$$0 < P_{ji}(t) = \frac{P_{ji}(t \cap t')}{P_{ji}(t'|t)} < 1 \ \forall \ t', \ t. \tag{1}$$

By (1), $P_{ji}(t'|t) > 0$ (and $0 < P_{ji}(t \cap t') < 1$). By induction, we have $P_{ji}(t'|t, t_1, ..., t_x) > 0 \ \forall \ t' > \ t, t_x$. Denote by $\mathsf{B}(t)$ the event that l_{ji} losses all messages (if any is sent) for the interval $t + \Delta t$, where Δt is the control cycle duration. Let t_x denote the times at which h_j sends a message in $[t + \Delta t]$. Then the probability of $\mathsf{B(t)}$ happening is:

$$\begin{aligned} Pr(\mathsf{B(t)}) &= P_{ji}(t_1 \cap t_2 \cap ... \cap t_m) \\ &= P_{ji}(t_1) \times P_{ji}(t_2|t_1) \times ... \times P_{ji}(t_m|t_1, ..., t_{m-1}) > 0. \end{aligned}$$

Given $0 < P_{ji}(t) < 1$ $\forall t$ and $P_{ji}(t'|t, t_1, ..., t_x) > 0$ $\forall \ t' > \ t, t_x$, then we have $0 < Pr(\mathsf{B(t)}) < 1$; there is a positive probability that h_i receives no reply from h_j and thus the invoked operation does not complete in any bounded cycle duration where a bounded number of messages can be sent by a host, which violates termination. ■

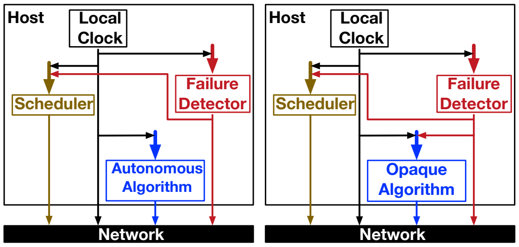

Figure 3. Different algorithmic families resembling traditional approaches for building distributed shared memory.

We now define and distinguish between different algorithmic families; these algorithmic families resemble traditional approaches for building distributed shared memory [1]–[5].

Autonomous Algorithms. These algorithms can only exchange messages using the lossy links of the DCS and use the available local synchronized clocks. In other words, these algorithms do not use any external building blocks (abstractions) but implement themselves all needed functionalities, thus the name autonomous.

The scheduler is oblivious to algorithms in this family; the scheduler does not adapt its behavior to the algorithm's decisions, i.e., changes to the algorithm result in no impact on the scheduler (Figure 3).

Opaque Algorithms. These algorithms can use, in addition to message exchange and local clocks, a failure detector as a black-box [23]. Roughly speaking, this means that opaque algorithms can observe the output of failure detectors and build on top of this output and the properties that this output satisfies (Figure 3). Different failure detector classes can be defined depending on the properties guaranteed by the output of the failure detector. Recall that we consider the output of a failure detector to be a list of suspected hosts.

The scheduler is not oblivious to the failure detector used, however, both the scheduler and the failure detector are oblivious to the algorithms in this family. It is important to note that algorithms in this family are not a subset of autonomous algorithms, since implementing the failure detector itself may not be possible via message exchange and local clocks.

Theorem 1. *No autonomous algorithm can deterministically implement termination and freshness in a DCS, where up to n-2 hosts can fail.*

Proof: By contradiction, assume the existence of an autonomous algorithm $\mathcal{A}$ that deterministically implements termination and freshness. $\mathcal{A}$ cannot know which hosts are seen as "non-suspected" by the scheduler and thus should guarantee freshness for any non-crashed host.

For illustration, we consider a DCS of two correct hosts h_1 and h_2. By Lemma 1, operations (reads and writes) satisfying termination in $\mathcal{A}$ complete without waiting for any reply from any host. Hosts in $\mathcal{A}$, however can exchange messages to ensure freshness. Consider host h_1 to be the writer of a shared object $\mathcal{O}$, i.e., h_1 invokes write operations to $\mathcal{O}$ at every cycle.

Similar to the reasoning in the proof of Lemma 1, we compute the probability that all messages sent from and to h_1 are lost for $\alpha \cdot c$ cycles $\forall\ \alpha \geq 1$.

Let $P_{21}(t \cap t')$ ($P_{12}(t \cap t')$) be the probability that l_{21} (l_{12}) loses the messages (if any is sent) at time t and time t'. $0 < P_{21}(t), P_{12}(t) < 1\ \forall t$, then:

$$0 < P_{21}(t) = \frac{P_{21}(t \cap t')}{P_{21}(t'|t)}, P_{12}(t) = \frac{P_{12}(t \cap t')}{P_{12}(t'|t)} < 1\ \forall\ t',\ t. \tag{2}$$

By (2), $P_{21}(t'|t) > 0$ ($0 < P_{21}(t \cap t') < 1$) and $P_{12}(t'|t) > 0$ ($0 < P_{12}(t \cap t') < 1$). By induction, we have $P_{21}(t'|t, t_1, ..., t_x) > 0$ and $P_{12}(t'|t, t_1, ..., t_x) > 0$ $\forall\ t' >\ t, t_x$. Denote by $\mathsf{R}(t)$ ($\mathsf{S}(t)$) the event that l_{21} (l_{12}) loses all messages (if any is sent) for the interval $t + \Delta t$, where Δt is such that $t + \Delta t$ is equal to the duration of $\alpha \cdot c$ cycles. Let m (m') be the maximum number of messages h_1 (h_2) can send in $t + \Delta t$ and t_x (t'_x) denote the times at which a message is sent from (to) h_1 during $[t, t + \Delta t]$. Then the probabilities of $\mathsf{R(t)}$ and $\mathsf{S(t)}$ happening is:

$$\begin{aligned} Pr(\mathsf{R(t)}) &= P_{21}(t_1 \cap t_2 \cap t_3 \cap ... \cap t_m) \\ &= P_{21}(t_1) \times P_{21}(t_2|t_1) \times ... \times P_{21}(t_m|t_1, ..., t_{m-1}) > 0. \end{aligned}$$

$$\begin{aligned} Pr(\mathsf{S(t)}) &= P_{12}(t'_1 \cap t'_2 \cap ... \cap t'_m) \\ &= P_{12}(t'_1) \times P_{12}(t'_2|t'_1) \times ... \times P_{12}(t'_m|t'_1, ..., t'_{m-1}) > 0. \end{aligned}$$

Given $0 < P_{21}(t),\ P_{12}(t) < 1$, $P_{21}(t'|t, t_1, ..., t_x) > 0$ and $P_{12}(t'|t, t_1, ..., t_x) > 0\ \forall\ t' >\ t, t_x$, then $0 <\ Pr(\mathsf{S(t)}), Pr(\mathsf{R(t)}) < 1$.

Consider some cycle r. The probability that all messages from and to h_1 get lost and thus all writes invoked by h_1 during cycles $[r, r + \alpha \cdot c]$ are not seen by h_2 is:

$$Pr(\mathsf{S(t)} \cap \mathsf{R(t)}) = Pr(\mathsf{S(t)}) \cdot Pr(\mathsf{R(t)}|\mathsf{S(t)}).$$

Since $0 < Pr(\mathsf{S(t)}) < 1$, then $0 < \frac{Pr(\mathsf{S(t)} \cap \mathsf{R(t)})}{Pr(\mathsf{R(t)}|\mathsf{S(t)})} < 1$ and $0 < Pr(\mathsf{S(t)} \cap \mathsf{R(t)}) < 1$.

Thus, h_2 invoking a read operation to read the value $\mathcal{O}$, for example at cycle $r + c + 1$, has a positive probability of reading a value that is not written c cycles ago, violating freshness. Recall that, by the termination property, every read operation should complete and return a value within the same cycle in which it was invoked. ■

We now define what we call a *non-trivial* failure detector.

Definition 1. *A non-trivial failure detector*[5] *does not suspect a correct host, at any point in time, with positive probability, while it eventually suspects all failed hosts permanently.*

Theorem 2. *Let $\nabla\mathcal{A}(s)$ denote the set of hosts which are not suspected by a non-trivial failure detector $\mathcal{X}$ during cycle s. No opaque algorithm using $\mathcal{X}$ can deterministically implement termination and freshness for hosts $\in \nabla\mathcal{A}(s)\ \forall s$, in a DCS, where n-2 hosts can fail.*

Proof: An opaque algorithm can observe the output of the failure detector $\mathcal{X}$ and hence can know $\nabla\mathcal{A}(s)$, the set of hosts that are not suspected by $\mathcal{X}$ during cycle s. Consider a host h_i to be the writer of a shared object $\mathcal{O}$, i.e., h_i invokes write operations to $\mathcal{O}$ at every cycle.

Consider an execution where h_i and some other host h_j (which requires to read the value of $\mathcal{O}$ after cycle $r + c$) are correct. Then from Definition 1, there is a positive probability that $\{h_i,\ h_j\} \in \nabla\mathcal{A}(s)$, $\forall s \in [r, r + \alpha c]$ at hosts h_i and h_j.

Similar to the proof of Theorem 1, it can be inferred that there is a positive probability that all writes invoked by h_i during $[r, r + \alpha c]$, are not seen by host h_j (all messages sent by h_i during $[r, r + \alpha c]$ can be lost with positive probability). As such, all reads invoked by h_j after cycle $r + c$ do not return a fresh value. This concludes the proof. ■

IV. TapeWorm: A DCS Shared Memory Algorithm

In this section, we demonstrate a way to circumvent the impossibilities shown in Section III. We present *TapeWorm*, our algorithm for implementing shared memory in DCSs. The main idea underlying *TapeWorm* is to benefit from monitoring messages (known as heartbeats) typically exchanged by the failure detector component of a DCS, precisely by attaching information to these messages. Recall from Section II-C that we consider failure detectors that detect host crashes in real time, based only on the exchanged heartbeat messages and time-outs. Accordingly, the below is a necessary condition for any such failure detector that detects crashed hosts in a delay less than d_t cycles after the crash ($\forall d_t$).

Real-time detection: *If a host h_i does not hear any of the heartbeats sent by host h_j during $[r - (d_t - 1), r - 1]$ (directly from h_j or indirectly via other hosts), h_i suspects h_j at the beginning of cycle r.*

This suspicion places h_j in the eliminated set of hosts $\nabla\mathcal{E}$. d_t is fixed and constitutes the real-time guarantee for detecting a crashed host in the DCS.

A. A Basic TapeWorm Algorithm

For simplicity, we describe *TapeWorm* (Algorithm 1) in the context of a single shared object $\mathcal{O}$. Recall that d_t is an upper bound on the number of cycles it takes a failed host to be declared as failed by the system (precisely suspected by the failure detector). We assume that $c > d_t$, c being the bound on the data freshness.

Every host in *TapeWorm* maintains a list, $Fresh_{list}$, of size c. $Fresh_{list}$ of a host h at cycle r holds the values of $\mathcal{O}$ seen by h and tagged with cycles in $[r - c, r]$. Later in Section IV-A, we show that it is sufficient for $Fresh_{list}$

[5] A failure detector providing more accurate information about correct hosts in the system is only a special case of a non-trivial failure detector.

Algorithm 1 A Basic *TapeWorm* Algorithm.

```
Initialize:
set $Fresh_{list} = \{\perp\}^c$

Repeat periodically:
broadcast $< \mathsf{heartbeat}, \mathsf{Fresh_{list}}, \mathsf{id} >$

upon event receive $< \mathsf{heartbeat}, \mathsf{Fresh'_{list}}, \mathsf{id'} >$ do
    set $Fresh_{list} = Fresh'_{list} \bigcup Fresh_{list}$

upon event write $< \mathsf{v}, \mathsf{cycle} >$ do
    $update_{list} < \mathsf{v}, \mathsf{cycle} >$

upon event read $< cycle >$ do
    if ($cycle < c$) then
        return $\perp$
    end if
    if ($< *, \mathsf{cyc} > \in \mathsf{Fresh_{list}}$ : $cyc = \max\{cyc \leq cycle - d_t\}$) then
        return $< *, \mathsf{cyc} >$
    end if
    return $< *, \mathsf{cyc} > \in \mathsf{Fresh_{list}}$ : $cyc = \max cyc$

Function $update\_list(< \mathsf{v}, \mathsf{cycle} >)$:
    set $Fresh_{list} = Fresh_{list} \bigcup < \mathsf{v}, \mathsf{cycle} >$
    remove $< *, cycle - c - 1 >$ from $Fresh_{list}$
```

to hold values of $\mathcal{O}$ seen by h and tagged with cycles in $[r - d_t, r]$.

Consider h_w to be the host that writes to $\mathcal{O}$ (recall that any object is written to by a single host). Upon invoking a write to object $\mathcal{O}$ with a value v at cycle r, h_w updates its $Fresh_{list}$, by adding the tuple $< r, v >$ and deleting the tuple tagged with control cycle $r - c - 1$. After this step, a write completes.

A host h that invokes a read operation to object $\mathcal{O}$ at cycle r, consults its $Fresh_{list}$. Host h returns the value tagged with the largest cycle number, say max_{cycle}, such that $max_{cycle} \leq r - d_t$. After this step the read invoked by h completes. If no such value exists, then a read returns the value tagged with the largest cycle within $Fresh_{list}$. Initially, for the first c cycles, reads return $\perp$, the initial value of $\mathcal{O}$ (assumed to be known by all hosts) and completes.

Every host h benefits from the underlying heartbeats sent in the DCS by piggybacking $Fresh_{list}(h)$ on these heartbeats. We consider that heartbeats are scheduled to be exchanged towards the end of a control cycle, i.e., after all invoked operations during a cycle have completed. A host h_i, upon receiving a heartbeat during cycle r from host h_j, updates its $Fresh_{list}(h_i)$ to:

$$Fresh_{list}(h_i) = Fresh_{list}(h_i) \bigcup Fresh_{list}(h_j),$$

for all values tagged with control cycles in $[r, r - c]$.

Proof of Correctness.

We now prove the correctness of our *TapeWorm* algorithm.

Termination: Both read and write operations in *TapeWorm* complete after performing a bounded number of local operations which requires a bounded amount of time and thus constitutes the required t_{op}. Termination is hence satisfied.

Agreement and freshness: Recall that it is sufficient to guarantee the shared memory properties when the scheduler state is consistent (Section II-D). The scheduler is consistent when all hosts $\in \nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)$ at any cycle r agree on which hosts are in $\nabla\mathcal{A}(r+1)$ during cycle $r+1$ (see Section II-C). Given that the scheduler state is consistent, we have the following:

Lemma 2. *For any host $h \in \nabla\mathcal{A}(r)$, i.e., belonging to the set of non-suspected hosts at cycle r, all hosts in $\{\nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)\}$ have heard (directly or indirectly) a heartbeat sent by h during some cycle in $[r-(d_t-1), r-1]$.*

Proof: By the real-time detection property, a crashed host is declared failed (suspected) in less than d_t cycles after the crash. Consider a DCS of two hosts h_1 and h_2 and consider the following two executions:

- $e1$: an execution where h_2 crashes during cycle $r - (d_t - 1)$ before h_2 sends any heartbeats.
- $e2$: an execution where h_1 and h_2 are both correct (do not crash) but lose all heartbeats sent (if any) during all cycles in $[r - (d_t - 1), r - 1]$.

Execution $e2$ can happen with positive probability as shown in the proof of Lemma 1. With respect to h_1, executions $e1$ and $e2$ are indistinguishable during $[r - (d_t - 1), r - 1]$, for any finite value of d_t (since h_1 cannot know if h_2 has failed or all messages from h_2 are lost).

By the real-time property of failure detection, h_1 suspects h_2 in execution $e1$ at cycle r. Since $e1$ and $e2$ are indistinguishable during $[r - (d_t - 1), r - 1]$ then h_1 suspects h_2 as failed at cycle r also in $e2$. So, in order for h_2 not to be suspected at cycle r by h_1, h_1 has to hear (directly or indirectly) at least one heartbeat sent by h_2 during some cycle in $[r - (d_t - 1), r - 1]$. Since the scheduler state is assumed to be consistent, then all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have heard (directly or indirectly) a heartbeat sent by host h_i, at some cycle in $[r - (d_t - 1), r - 1]$, $\forall h_i \in \nabla\mathcal{A}(r)$. ■

Lemma 3. *Let h be the host that writes to the shared object $\mathcal{O}$. If $h \in \nabla\mathcal{A}(r)$, then all hosts $\in \nabla\mathcal{E}(r - 1) \bigcup \nabla\mathcal{A}(r-1)$ have in their $Fresh_{list}$ the value written by h at cycle $r - d_t$.*

Proof: By Lemma 2, if $h \in \nabla\mathcal{A}(r)$, then all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have heard (directly or indirectly) a heartbeat sent by host h at some cycle in $[r-(d_t-1), r-1]$. All heartbeats sent by h at some cycle in $[r-(d_t-1), r-1]$ contain the value written by h at cycle $r - d_t$, since (i) heartbeats are exchanged towards the end of a cycle (after operation in that cycle have completed) and (ii) h always appends to its heartbeats $Fresh_{list}(h)$, which contains all values written by h in cycles $[r - c, r]$, where $d_t < c$. In fact, it can be noticed that it is sufficient to only send $Fresh_{list}(h)$ containing values of $\mathcal{O}$ written by h in cycles $[r - d_t, r]$.

Since every host $\in \nabla\mathcal{E}(r - 1) \bigcup \nabla\mathcal{A}(r - 1)$ has heard (directly or indirectly) a heartbeat sent by host $h \in \nabla\mathcal{A}(r)$ at some cycle in $[r-(d_t-1), r-1]$, then every host has received from some host h_i a $Fresh_{list}(h_i)$ containing the value of object $\mathcal{O}$ written at cycle $r - d_t$. The reason is that any host h_j receiving $Fresh_{list}(h_i)$ performs: $Fresh_{list}(h_j) = Fresh_{list}(h_i) \bigcup Fresh_{list}(h_j)$, concluding the proof. ■

By Lemma 3 and the algorithm description, if h, the writer host of some object $\mathcal{O}$ is not suspected at cycle r, then all hosts invoking a read to $\mathcal{O}$ during r return the value written at cycle $r - d_t$, which satisfies agreement and freshness since $c > d_t$.

If, however, h is suspected as crashed at cycle r, then by Lemma 2 and Lemma 3 all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have last heard (directly or indirectly), the heartbeat sent by h during cycle $r - d_t$ (otherwise h would suspected at a cycle $< r$). This heartbeat contains the value written by h at cycle $r - d_t$. No other heartbeats are heard from h (since otherwise h is suspected at cycle $> r$). The value tagged with $r - d_t$ hence has the highest cycle and is thus returned by any host issuing reads during cycles $\geq r$, which satisfies both agreement and freshness, as $c > d_t$. *TapeWorm* hence guarantees agreement and freshness when h is not suspected and when h is. Note that from the proof of Lemma 3, it is sufficient that $Fresh_{list}$ of hosts is of size d_t and not c.

B. A Fresher TapeWorm Algorithm

In this section, we depict how *TapeWorm* can be modified such that read operations return fresher values, within the defined freshness interval. In other words, we describe how reads invoked in *TapeWorm* at cycle r can return values written at cycles in $[r - c, r]$, precisely in $[r - s, r]$ where $s < d_t$.

Every host in *TapeWorm* has a list, $Fresh_{list}$, that holds at cycle r the values of $\mathcal{O}$ seen by h and tagged with control cycles in $[r - c, r]$. For every value in $Fresh_{list}(h)$, h now keeps a list of host ids, called $seen_{cycle\#}$, and for each host id in $seen_{cycle\#}$ a list called $seenby_{id}$.

Upon invoking a write, with a value v, to object $\mathcal{O}$ at cycle r, h_w (the host writing to $\mathcal{O}$) appends its id to $seen_r$, besides updating its $Fresh_{list}$ as described in Section IV-A. After this step, a write completes.

Every host h piggybacks its $Fresh_{list}(h)$, $seen_{cycle\#}$ and $seenby_{id}$ lists to the heartbeats. Upon receiving a heartbeat at cycle r from host h_j, a host h_i updates its $Fresh_{list}(h_i)$ to:

$$Fresh_{list}(h_i) = Fresh_{list}(h_i) \bigcup Fresh_{list}(h_j),$$

for all values tagged with control cycles in $[r, r - c]$. For every value tagged with cycle r' such that r' is in $Fresh_{list}(h_j)$ but not in $Fresh_{list}(h_i)$, h_i adds its id to $seen_{r'}(h_i)$. Afterwards, for every value tagged with $cycle\#$ in the newly computed $Fresh_{list}(h_i)$, h_i performs:

$$seen_{cycle\#}(h_i) = seen_{cycle\#}(h_i) \bigcup seen_{cycle\#}(h_j),$$

and for every host h_k ($k \neq j$) in the new $seen_{cycle\#}(h_i)$:

$$seenby_{h_k}(h_i) = seenby_{h_k}(h_i) \bigcup seenby_{h_k}(h_j).$$

For h_j in the new $seen_{cycle\#}(h_i)$, h_i performs:

$$seenby_{h_j}(h_i) = seenby_{h_j}(h_i) \bigcup seen_{cycle\#}(h_j).$$

A host h that invokes a read operation to object $\mathcal{O}$ at cycle r, consults its $Fresh_{list}(h)$ and returns the value tagged with the largest cycle, $max_{cycle} > r - d_t$, satisfying both properties below:

1) $seen_{max_{cycle}}$ contains the ids of all hosts that are not suspected in cycle $r + 1$.
2) For every host h_k in $seen_{max_{cycle}}$, $seenby_{h_k}(h)$ contains the ids of all hosts not suspected in $r + 1$.

The above two properties state that a host h returns a value at cycle r, if h knows that (i) every host in $\nabla\mathcal{A}(r + 1)$ (non-suspected hosts at cycle $r + 1$) has seen that value and (ii) every host in $\nabla\mathcal{A}(r + 1)$ knows that every other host in $\nabla\mathcal{A}(r + 1)$ has seen that value. After this step the read invoked by h completes. If no such value exists, then a read returns as it would do in the basic *TapeWorm* algorithm described in Section IV-A. As such, the correctness of this fresher *TapeWorm* follows from that of the basic version.

V. Performance Analysis

In this section, we analyze certain aspects of *TapeWorm*'s performance. We specifically determine (a) the probability distribution of the values returned by read operations in the allowable freshness range, i.e., $[cycle_{read} - c, cycle_{read}]$ where $cycle_{read}$ is the cycle in which a read operation is invoked, (b) the bandwidth overhead of *TapeWorm*, and (c) optimizations of *TapeWorm* for static workloads.

A. The Freshness of Values Seen by Hosts

Recall that heartbeats, to which *TapeWorm* attaches information, are exchanged at the end of a cycle, after all tasks are executed. In other words, read and write operations get invoked and complete at a cycle r before the heartbeats at cycle r get exchanged. For simplicity, we consider $P_{ij}(t) = p$ $\forall i, j$ and t. In other words, a message sent at any time and on any link has probability p of getting lost, where p is the same for all links and is independent of time and links. We study the fresher *TapeWorm* version depicted in Section IV-B assuming that the writer host is correct, i.e., does not crash.

Assume that for all cycles in $[r + 1, r + d_t]$:

$$\nabla\mathcal{A}(r + 1) = \nabla\mathcal{A}(r + 2) = ... = \nabla\mathcal{A}(r + d_t) = \nabla\mathcal{A},$$

i.e., the set of non-suspected hosts remains the same. We now compute the probability that a read, in *TapeWorm*, to a shared object $\mathcal{O}$ at any cycle in $[r+1, r+d_t]$ returns the value of $\mathcal{O}$ written at cycle r. In the fresher *TapeWorm* version, a read to $\mathcal{O}$ invoked at cycle $r+1$, returns the value written at cycle r with probability 0. This is due to the following fact: a host h sends its heartbeats at cycle s before it receives any heartbeats that some other hosts sent during s.

A read to $\mathcal{O}$ at cycle $r + 2$, returns the value written at cycle r with probability $(1 - p)^{|\nabla\mathcal{A}|^2(|\nabla\mathcal{A}|-1)}$.

For reads invoked at cycle $cyc \in [r+3, r+d_t-1]$: the probability that a read at cycle cyc returns the value of $\mathcal{O}$ written at cycle r can be approximated by the probability that each host in $\nabla\mathcal{A}$ hears (directly or indirectly) from every other host in $\nabla\mathcal{A}$ by cycle $cyc-2$, after which every host in $\nabla\mathcal{A}$ hears the heartbeat of every other host in $\nabla\mathcal{A}$. Let h be a host in $\nabla\mathcal{A}$. We denote by $\pi_h(cyc)$ the set of hosts in $\nabla\mathcal{A}$ that received a heartbeat from h (directly or indirectly) at some cycle in $[r+3, r+cyc-3]$ and by $\pi_{\bar{h}}(cyc)$ the set of hosts in $\nabla\mathcal{A}$ that did not receive a heartbeat from h (directly or indirectly) at some cycle in $[r+3, r+cyc-3]$. The probability that at cycle $cyc-2$ not all hosts in $\nabla\mathcal{A}$ hear from h is equal to the probability that at least one host in $\pi_{\bar{h}}(cyc)$ does not receive a heartbeat from any host in $\pi_h(cyc)$ in cycle $cyc-2$:

$$Prob(|\pi_{\bar{h}}(cyc)|)\times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x},$$

where $Prob(\pi_{\bar{h}}(cyc))$ is the probability that $|\pi_{\bar{h}}(cyc)|$ hosts do not hear (directly or indirectly) from host h any heartbeat by cycle $cyc-3$. Thus, the probability that a read at cycle cyc returns the value of $\mathcal{O}$ written at cycle r is:

$$(1-p)^{|\nabla\mathcal{A}|} \times \prod_{\forall h \in \nabla\mathcal{A}} 1 - Prob(|\pi_h(cyc)|)\times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x}.$$

A read to $\mathcal{O}$ invoked at cycle $r+d_t$, returns the value written at cycle r with probability:

$$1-(1-p)^{|\nabla\mathcal{A}|(|\nabla\mathcal{A}|-1)} - \sum_{cyc=r+3}^{r-d_t-1} (1-p)^{|\nabla\mathcal{A}|} \prod_{\forall h \in \nabla\mathcal{A}} 1 - Prob(|\pi_h(cyc)|)\times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x}.$$

B. Bandwidth overhead of TapeWorm

Hosts in *TapeWorm* do not send additional messages. However, hosts append information to heartbeats. In this section, we quantify the size of information being piggybacked to heartbeats. Every host in *TapeWorm* saves c values of object $\mathcal{O}$ relative to the last c values written to $\mathcal{O}$. In fact, it is sufficient for hosts to keep the last $d_t < c$ values of $\mathcal{O}$, as shown in Section IV-A. These values constitute the $Fresh_{list}$.

Consider that shared memory consists of x shared objects each capable of storing a value of m bits. The basic *TapeWorm* algorithm of Section IV-A incurs a bandwidth overhead of $d_t \cdot\ x \cdot m$ bits/cycle per link. This overhead is relative to having each host attach the $Fresh_{list}$ to the heartbeat sent by that host every cycle.

In the fresher *TapeWorm* algorithm of Section IV-B a host sends, in addition to the $Fresh_{list}$, the $seen_{cycle\#}$ and $seenby_{id}$ lists. The information in these two lists can be represented by an $n\ \times\ (n+\ 1)$ binary matrix, where n is the total number of hosts in the system. The first column of this matrix represents information relative to the $seen_{cycle\#}$ list, and each row, excluding the first position, represents the information relative to $seenby_{id}$ list. A compact way of representing this matrix is to traverse the bits column by column (or row by row) and represent the binary data in ASCII format. The result incurs an additional overhead of $\frac{n(n+1)}{8}$ bits/cycle, per link compared to the basic *TapeWorm.*

Existing known failure detection and membership algorithms in control systems and real-time environments already implement an all-to-all broadcast mechanism for sending heartbeats and monitoring hosts [11], [12], [14], [25]–[27]. Often, these systems use Ethernet packets over UDP [11], [12], [25], [26] to send heartbeats. Classic heartbeats occupy only a very small fraction of the allowable minimum payload of an Ethernet packet (minimum UDP payload is 18 bytes).

Since TapeWorm only appends information to heartbeats, part of or maybe all information relative to TapeWorm (depending on the system and shared storage size) can be sent without any additional overhead by utilizing the unused payload of heartbeat messages.

C. DCS Shared Memory: Necessary & Sufficient Conditions

In this section, we determine the necessary and sufficient conditions for implementing the three properties of shared memory in a DCS (defined in Section II-D).

We recall sufficient assumptions that define the family of algorithms to which *TapeWorm* belongs; we refer to this family of algorithms as *Parasite Algorithms*:

1) Hosts can exchange messages or append information to heartbeats exchanged over the DCS links. Precisely, in every cycle, each host either appends or does not append information to the heartbeat broadcasted in that cycle. Hosts have access to local synchronized clocks.
2) Hosts get suspected according to the real-time detection property: if a host h_i does not hear any of the heartbeats sent by host h_j during $[r-1, r-(d_t-1)]$ (directly from h_j or indirectly via other hosts), h_i suspects h_j at the beginning of cycle r.

Theorem 3. *A necessary condition for a parasite algorithm to deterministically implement termination, agreement and freshness in a DCS is:*

Every host h appends, to every heartbeat sent in $[r-\ 1, r-\ (d_t-1)]$, any value of object $\mathcal{O}$ written during a cycle $\in [r-\ c, r]$, if h knows of any such value (where $r+1$ is the cycle during which a read operation is invoked by some host).

Proof: Let $r+1$ be the cycle at which a read to object $\mathcal{O}$ is invoked by some host. To prove Theorem 3, we prove that if some host h, be it a writer or a reader, knows a value of $\mathcal{O}$ written during some cycle $\in [r-\ c, r]$ and does not append any such value to some heartbeat sent in $[r-1, r-(d_t-1)]$, one of the three properties is violated.

Consider a DCS with three hosts h_1, h_2 and h_3 sharing an object $\mathcal{O}$, where h_1 is the writer of $\mathcal{O}$. Also assume that at cycle $r+1=c+1$, both h_2 and h_3 invoke a read to $\mathcal{O}$. To satisfy agreement and freshness, these reads should return the same value, that being a value written earliest at cycle 1.

For illustration consider $c = d_t + 1$. In this case, since $r+1=c+1$, a host h by Theorem 3 should append values of $\mathcal{O}$ to every heartbeat sent at all cycles $\in [3, c]$.

Case 1: h is the writer.

Assume that h_1 decides not to append any information to the heartbeat it broadcasts at some cycle $r' \in [3, c]$. Consider an execution e satisfying all the below:

1) All three hosts are correct, i.e., no host fails.
2) Both h_2 and h_3 lose all heartbeats sent by h_1 at all cycles in $[1, r'[\ \bigcup\ ]r', c]$. However, h_2 and h_3 both receive the heartbeat sent by h_1 at r'.
3) h_1 and h_2 receive all the heartbeats sent by h_3 during all cycles in $[1, c]$.
4) h_1 and h_3 receive all the heartbeats sent by h_2 during all cycles in $[1, c]$.

Execution e can happen with positive probability (since each host broadcasts a heartbeat at every cycle and every sent heartbeat has a positive probability of being lost). In execution e, the failure detector at the beginning of cycle $c+1$, at all hosts, includes h_1, h_2 and h_3 in $\nabla\mathcal{A}$, since every host heard some heartbeat sent from every other host during the past $d_t - 1$ cycles, i.e., in $[3, c]$. However, h_2 and h_3 do not see any value for $\mathcal{O}$ besides $\perp$, the initial value (prior to any write); h_2 and h_3 only receive the heartbeat sent by h_1 at cycle r' and this heartbeat has no values of $\mathcal{O}$ appended to it. As such, a read at cycle $c+1$ invoked by either h_2 or h_3 and satisfying termination completes and returns $\perp$ during $c+1$, which violates the freshness property.

Case 2: h is a reader.

Assume now that h_2 does not append any value to the heartbeat it broadcasts at some cycle $r' \in\ [3, c]$. Consider an execution e' satisfying all the below:

1) All three hosts are correct.
2) h_3 loses all heartbeats sent by h_1 at all cycles in $[1, c]$.
3) h_2 receives all heartbeats sent by h_1, and h_1 receives all heartbeats sent by h_2 at all cycles in $[1, c]$.
4) h_3 loses the heartbeats sent by h_2 at all cycles in $[1, r'[\ \bigcup\ ]r', c]$, but receives the heartbeat sent by h_2 at cycle $r' \in [3, c]$.
5) h_1 and h_2 receive all the heartbeats sent by h_3 at all cycles in $[1, c]$.

Execution e' can happen with positive probability (since messages can be probabilistically lost). In e', the failure detector at the beginning of cycle $c+1$, at all hosts, can include h_1, h_2 and h_3 in $\nabla\mathcal{A}$, since every host can hear (directly or indirectly) some heartbeat sent from every other host during the past $d_t - 1$ cycles, i.e., during $[3, c]$. Specifically h_3 can hear the heartbeat of h_1 indirectly via the heartbeat received from h_2 at cycle r'.

However, since h_2 did not append any value for $\mathcal{O}$ at cycle r', h_3 does not see any value for $\mathcal{O}$ besides the initial value $\perp$. Note that h_2 knows values of $\mathcal{O}$ since it receives heartbeats from h_1 (h_1 is the writer of object $\mathcal{O}$ and appends values of $\mathcal{O}$ to all the sent heartbeats). As such, a read at cycle $c+1$ invoked by h_3 and satisfying termination completes and returns $\perp$ during $c+1$, which violates the freshness property. ■

Theorem 4. *Consider that non-crashed hosts agree on the set $\nabla\mathcal{A}(r)$, such that the writer of a shared object $\mathcal{O}$ belongs to $\nabla\mathcal{A}(r)$. Then a sufficient condition for a parasite algorithm to deterministically ensure termination, agreement and freshness in a DCS, given a single object $\mathcal{O}$ to which a read operation is invoked by some host at cycle r, is:*

Each host appends any value of the shared object written at some cycle in $[r-c, r-d_t]$ (if this host has seen such a value) to every heartbeat sent during all cycles in $[r-d_t, r]$.

Proof: Consider a DCS of three hosts h_1, h_2 and h_3. Also consider h_1 to be the host that writes to $\mathcal{O}$.

Let r be the cycle at which some host invokes a read to $\mathcal{O}$ and let v denote the value that h_1 writes to $\mathcal{O}$ at cycle $r - d_t$.

Lemma 4. *Consider that every host appends v (whenever it has received v) to every heartbeat sent during all cycles in $[r-d_t, r]$, where r is the cycle during which some host invokes a read operation to that shared object. Given that hosts agree on the set $\nabla\mathcal{A}(r)$, all non-crashed hosts at cycle r have v.*

Proof: Upon receiving a write to object $\mathcal{O}$ at cycle $r - d_t$, h_1 saves v and then completes, i.e., the write at h_1 locally returns before h_1 sends any heartbeat at $r - d_t$. h_1 hence appends v to all the heartbeats it sends in cycles $\in [r-d_t, r]$ (since h_1 has v). Agreeing about $\nabla\mathcal{A}(r)$, where $h_1 \in \nabla\mathcal{A}(r)$, means that all hosts in $\{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)\}$ have heard (directly or indirectly) a heartbeat sent by h_1 during some cycle in $[r-d_t, r]$. Every host that hears v, appends v to every heartbeat it sends in $[r-d_t, r]$. This implies that every host in $\{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)\}$, which was able to hear (directly or indirectly) a heartbeat from h_1, has received the value v. ■

By Lemma 4 all non-crashed hosts at cycle r would have received v. Any host that invokes a read at r can thus always complete after locally returning v, the value of $\mathcal{O}$ written at cycle $r - d_t$, deterministically. Readers and writers locally return satisfying termination. All readers at cycle r return v, thus satisfying agreement and freshness (since $c > d_t$). ■

D. Optimizing TapeWorm under Static Workloads

We have assumed, so far, that read operations can be invoked at any cycle and this invocation time in not known. As such, in both versions of *TapeWorm*, the basic and the fresher, information about a shared object is appended to heartbeats of every host at every cycle. Based on the results of Section V-C, we investigate optimizing (i) the rate of appending information to heartbeats and (ii) the number of hosts that need to append information to heartbeats in every cycle.

When workloads are static, i.e., hosts know the cycle at which read operations are invoked, then *TapeWorm* under certain assumptions can be optimized, in the sense that hosts do not need to append information on all heartbeats sent at all control cycles. Precisely, from Theorem 3 and Theorem 4, satisfying the three properties of shared memory requires two things: (i) all non-faulty hosts append information on every heartbeat only for a fixed time interval (d_t cycles) before the invocation of a read operation, and (ii) the writer host does not get suspected during that interval of d_t cycles.

This can be interpreted as follows. Let r be the cycle at which some host invokes a read operation to object $\mathcal{O}$. In static workloads, r is known by *TapeWorm*. Hosts in *TapeWorm* can now append information to heartbeats sent only during d_t cycles prior to r. This optimization is valid under the assumption that the writer host of the object $\mathcal{O}$ can communicate (directly or indirectly) with all non-crashed hosts during that interval.

VI. Related Work

Sharing memory in real time has been addressed in various contexts, ranging from architectural design and synchronization for real-time shared memories in multi-core machines [34], [35] to real-time replicated databases [7]–[9] and distributed memory. In this section, however, we focus on previous related work in distributed environments (similar to DCSs) rather than on works (i) on architectural memory designs or (ii) on accessing physical shared memory in multi-core machines in real time.

Aslinger and Son [9] presented two algorithms for database replication, each targeting a different data workload. The first algorithm is developed for non-static periodic workloads and ensures that replicas are updated at the minimum required rate. The second algorithm is designed for random workloads and adaptively changes the update policy of replicas based on previously observed data patterns. Both algorithms aim at increasing the scalability of real-time databases. Peddi and DiPippo [8] proposed a database replication algorithm for static periodic workloads, where all object locations and client data requirements are known a priori. The algorithm creates transactions from the operations issued to the database objects and feeds these transactions to a scheduling algorithm. If a schedule that meets all deadlines can be computed, then all read operations guarantee to return a fresh value (satisfying temporal consistency). Otherwise, the system specification must be reconsidered. Wei et al. [7] use a full replication mechanism to ensure data freshness of committed transactions in medium distributed databases (5 to 10 nodes). The algorithm consists of local heuristic feedback controllers and global load balancers. The local controllers manage the admission of incoming workloads, while the global balancers collect performance data from all sites and balance the workloads.

In contrast to [7]–[9], in our paper we consider message losses in addition to node failures. Overcoming the effect of message losses and meeting the requirements of shared memory in a DCS is highly non-trivial, as we show that it is impossible to be achieved (Section III) without the aid of heartbeats and without requiring real-time crash detection.

Zou and Farnam [10] presented a real-time primary-backup replication scheme. The proposed scheme enforces *temporal* consistency (defined in Section II-D) among data replicas and determines the corresponding rate at which update messages should be sent from the primary to the backup. Zou and Farnam [10] also discussed message losses. The authors assumed that messages can be lost with probability ρ, and denoted by P the probability of the temporal consistency desired to be achieved. Their solution to message losses, as such, dictates to increase the frequency of sending update messages from the primary to the backup to guarantee the required probability P.

In contrast to [10], we seek in our paper to deterministically guarantee the consistency of operations issued to a shared object, given system agreement regarding which hosts are considered alive. Recall that those alive hosts are the ones that participate in executing tasks.

Xiong at al. [36] proposed MIRROR, a concurrency control algorithm for real-time replication control. MIRROR augments the optimistic two-phase locking (O2PL) algorithm with a state-based conflict resolution mechanism. The choice of the conflict resolution method is a dynamic function that either uses *Priority Abort* or *Priority Blocking* depending on the states of the transactions involved in the conflict. In Priority Abort, a conflict is resolved for the favor of the transaction with higher priority (by aborting a lower priority transaction currently holding the lock or blocking the lower priority transaction trying to acquire a lock held by a higher priority transaction). In Priority Blocking, a transaction is always blocked upon the encounter of a lock conflict and can acquire the lock after the lock is released. Lock requests are ordered by transaction priority.

Concurrency control mechanisms, such as [36], suffer from potential deadlock or unbounded blocking and thus do not comply with DCS-like requirements.

Other approaches addressed building real-time distributed hash tables (DHTs) [6], [37]. Qian et al. [6] designed and implemented a Chord-based DHT. Given the periodic structure of requests considered in [6], a cyclic executive is used to schedule the jobs that subsequent nodes in the Chord overlay network should execute to serve a request. Skodzik et al. [37] extended Kad, an implementation variant of the P2P Kademlia protocol, by a TDMA based mechanism in order to make Kad suitable for hard real-time constraints.

In contrast to [6], [37], we require each operation, be it a read or a write, to return and complete based on performing a bounded number of local steps. As we show in Lemma 1, waiting to reliably transmit any message in order for an operation to complete, could take an arbitrary long time in the communication system we consider in this paper.

VII. Conclusion

This paper investigated how to build a shared memory abstraction for distributed control systems (DCSs). Such an abstraction constitutes a basic building block for real-time shared storage functionalities (like real-time DHTs, key-value stores etc.), which are highly demanded in DCSs. We determined the guarantees that a shared memory abstraction should deliver to applications accessing it via read and write operations. We proved that such guarantees are impossible to implement deterministically, in the presence of host crashes and message losses (in the sense described in Section III). We presented *TapeWorm*, an algorithm that circumvents this impossibility and guarantees the desired shared memory properties for applications. *TapeWorm* adopts a white-box approach in which heartbeat messages of the failure detector component running in a DCS, are used as a means of transporting information. We also conducted a mathematical analysis quantifying the performance of *TapeWorm* and showcased ways for adapting and optimizing *TapeWorm* to application needs and workloads.

References

[1] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[2] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer-Verlag New York, 2011.

[3] M. Vukolic, *Quorum Systems:With Applications to Storage and Consensus*. Morgan and Claypool, 2012.

[4] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, 1995.

[5] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty, "How fast can a distributed atomic read be?" in *PODC*, 2004.

[6] T. Qian, F. Mueller, and Y. Xin, "A real-time distributed hash table," in *RTCSA*, 2014.

[7] Y. Wei, S. H. Son, J. A. Stankovic, and K. D. Kang, "Qos management in replicated real-time databases," in *RTSS*, 2003.

[8] P. Peddi and L. C. DiPippo, "A replication strategy for distributed real-time object-oriented databases," in *ISORC*, 2002.

[9] A. Aslinger and S. H. Son, "Efficient replication control in distributed real-time databases," in *AICCSA*, 2005.

[10] H. Zou and F. Jahanian, "A real-time primary-backup replication service," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, 1999.

[11] M. Oriol, T. Gamer, T. de Gooijer, M. Wahler, and E. Ferranti, "Fault-tolerant fault tolerance for component-based automation systems," in *ISARCS*, 2013.

[12] H. Kopetz and G. Grunsteidl, "Ttp - a time-triggered protocol for fault-tolerant real-time systems," in *FTCS*, 1993.

[13] E. Latronico and P. Koopman, "Design time reliability analysis of distributed fault tolerance algorithms," in *DSN*, 2005.

[14] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, "Safer: System-level architecture for failure evasion in real-time applications," in *RTSS*, 2012.

[15] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," in *PRDC*, 2001.

[16] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *DSN*, 2002.

[17] N. Hayashibara, X. Defago, and T. Katayama, "Two-ways adaptive failure detection with the phi-failure detector," in *ICAC*, 2003.

[18] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *PODC*, 2001.

[19] M. Felser, "Real-time ethernet - industry prospective," *Proceedings of the IEEE*, vol. 93, 2005.

[20] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti, "A hierarchical framework for component-based real-time systems," in *LNCS*, 2004, vol. 3054.

[21] G. Lipari, E. Bini, and G. Folher, "A framework for composing real-time schedulers," *ENTCS*, vol. 82, 2003.

[22] D. Dzung, R. Guerraoui, D. Kozhaya, and Y.-A. Pignolet, "To transmit now or not to transmit now," in *SRDS*, 2015.

[23] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, 1996.

[24] M. Larrea, A. Fernandez, and S. Arevalo, "Optimal implementation of the weakest failure detector for solving consensus," in *SRDS*, 2000.

[25] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "Rtcast: lightweight multicast for real-time process groups," in *RTTAS*, 1996.

[26] R. Barbosa, A. Ferreira, and J. Karlsson, "Implementation of a flexible membership protocol on a real-time ethernet prototype," in *PRDC*, 2007.

[27] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," *ACM Trans. Comput. Syst.*, vol. 13, 1995.

[28] D. Dzung, R. Guerraoui, D. Kozhaya, and Y.-A. Pignolet, "Never say never - probabilistic and temporal failure detectors," in *IPDPS*, 2016.

[29] R. Guerraoui, D. Kozhaya, M. Oriol, and Y.-A. Pignolet, "Who's on board? probabilistic membership for real-time distributed control systems," in *SRDS*, 2016.

[30] M. Oriol, M. Wahler, R. Steiger, S. Stoeter, E. Vardar, H. Koziolek, and A. Kumar, "Fasa: A scalable software framework for distributed control systems," in *ISARCS*, 2012.

[31] A. Timbus, A. Oudalov, and C. N. M. Ho, "Islanding detection in smart grids," in *ECCE*, 2010.

[32] B. Galloway and G. P. Hancke, "Introduction to industrial control networks," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, 2013.

[33] F. B. Schneider, D. Gries, and R. D. Schlichting, "Fault-tolerant broadcasts," *Sci. Comput. Program.*, vol. 4, 1984.

[34] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems," in *DATE*, 2015.

[35] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, 2011.

[36] M. Xiong, K. Ramamritham, J. Haritsa, and J. A. Stankovic, "Mirror: a state-conscious concurrency control protocol for replicated real-time databases," in *RTAS*, 1999.

[37] J. Skodzik, P. Danielis, V. Altmann, and D. Timmermann, "Hartkad: A hard real-time kademlia approach," in *CCNC*, 2014.