Active Replication for Centrally Coordinated Teams of Autonomous Vehicles

Nasos Grigoropoulos, Manos Koutsoubelias and Spyros Lalis
University of Thessaly
Volos, Greece
Email: {athgrigo,emkouts,lalis}@uth.gr

Abstract—**Autonomous vehicles, drones in particular, are used to support a wide range of sensing and actuating missions. While these missions are typically coordinated by a human operator, it is attractive to automate this coordination through a computer program that retrieves information from the vehicles and issues commands to them according to the mission objectives. However, the fact that such a computer-driven system may interact with and affect the physical environment in a direct way, introduces several challenges. In particular, it is important to tolerate failures of the mission control computer as smoothly as possible, avoiding roll-backs that might lead to inconsistencies. To address this problem, we propose an active replication approach, ensuring that as long as at least one replica of the mission controller remains operational, the mission will progress in a consistent way and with full transparency for the mission program. We define the properties that should be satisfied to achieve the required consistency, and present system-level mechanisms that support both deterministic and non-deterministic mission programs. We then discuss a concrete implementation of the proposed approach for an existing programming framework targeting multi-drone applications. Finally, we give an analytical cost model for the communication overhead of the proposed approach, and report the actual execution delay incurred in our prototype implementation for indicative scenarios using a suitable simulation environment.**

I. Introduction

Context & motivation. The impressive developments in sensors, control systems and embedded computing, have given rise to autonomous unmanned vehicles (UVs), such as drones, with substantial navigation and obstacle avoidance capability. As a result, several inexpensive platforms that can be controlled simply via high-level commands are now available.

In several application domains, such as agriculture, surveying/mapping and surveillance [1], the benefits of using such platforms can become even greater when multiple UVs are employed, in a coordinated way, to perform a mission as a team. The currently established approach, which imposes a one-to-one relation between a human operator and a UV, is impractical, does not scale and limits the degree to which one can exploit the full potential of these autonomous platforms. Thus, there is great promise in fully automating missions that involve multiple UVs, by letting the mission be coordinated by a computer program, rather than human operators.

While a certain degree of self-organization can be achieved using swarming techniques, these typically assume a large number of homogeneous UVs with relatively limited sensing and processing capabilities [2]. However, several real-world applications can be efficiently supported by employing small teams consisting of a few but more powerful UVs with possibly different sensing/actuation capabilities. In this case, it can be more appropriate to coordinate the team via a centralized mission control program.

The problem. Centralized solutions offer many advantages in terms of programmability and management of the available sensing/actuation resources. But it is crucial for the system to be able to tolerate failures of the mission control computer, which is the single most important component for the continuity of the mission. This can be achieved using replication [3], so that the mission continues in a smooth way as long as at least one replica of the mission controller remains operational.

One option is to follow a passive replication approach using a primary-backup scheme [4], whereby the primary replica of the mission controller actively executes the mission program and periodically takes checkpoints and sends them to the backup replicas. However, taking checkpoints frequently can introduce significant delays in the execution of the mission, whereas if checkpoints are taken sporadically, in case the primary replica fails, the backup that takes its place in order to resume the mission could roll-back to a much older state, leading to consistency issues.

As an alternative option, one may adopt an active replication approach [5], where all the replicas of the mission controller execute the mission program concurrently to each other. This way the failure of a replica can be masked without resorting to roll-backs. But this approach also has several issues that need to be addressed in order to achieve the desired correct operation and transparent fault-tolerance.

Contribution. In this paper, we investigate how to support the active replication of the mission controller, while addressing all the related correctness/consistency issues. The main contributions are: (i) we present an approach for supporting the active replication of the mission controller in coordinated teams of autonomous vehicles; (ii) we define the properties that should be satisfied in order to ensure consistency, and highlight the issues that need to be addressed; (iii) we present, in detail, a system-level mechanism that tackles the problem for deterministic mission programs, and briefly discuss how to deal with non-deterministic mission programs; (iv) we capture the overheads of a concrete implementation of the proposed solution, analytically as well as experimentally.

The rest of the paper is organized as follows. Section II describes the system model we use as a baseline for our work. Section III defines the desired consistency properties, identifies the issues that arise when employing an active replication approach for the mission controller, and presents solutions for deterministic and non-deterministic mission programs. Section IV discusses a concrete implementation of the proposed solutions for a programming environment that supports multi-drone applications. Section V gives an analytical cost model for the communication overhead of the proposed approach, and presents the results of indicative tests that have been performed using our prototype implementation on top of a suitable simulation environment. Section VI gives an overview of related work. Finally, Section VII concludes the paper.

II. System Model

We view each autonomous vehicle as a sensor/actuator *node* with movement capability. Nodes are coordinated by a distinguished entity, the *mission controller*, which runs the mission logic. The mission controller acts as a master: it collects information from the nodes, takes coordination decisions, and issues high-level commands to the nodes according to the mission objectives. The nodes act as slaves, following the commands of the master.

In the spirit of service-oriented computing [6], the sensing, actuation and mobility/navigation capabilities of the nodes are exposed in a structured way, through one or more service calls. Some calls are used to issue commands to the nodes, others to retrieve information. The mission logic comes in the form of a program that runs in the mission controller and decides which service to invoke at any given point in time.

The interaction between the mission program and the node services is implemented in a transparent way, through suitable middleware/runtime support. Among other things, the middleware/runtime is responsible for implementing the remote service invocation in the spirit of remote procedure calls (RPCs) [7]. Each service call translates to a corresponding request that is sent to the target node. In turn, the node processes the request and sends back the reply. This can be supported using suitable RPC support, or be implemented directly on top of a reliable transport service, such as TCP/IP.

For node failures, we assume the fail-stop model: a node either functions properly or stops. This is according to standard practice for embedded systems, which internally employ redundancy and/or secure state estimation techniques [8] [9] so that they are at least able to detect software/hardware failures of sensors/actuators as well as adversarial attacks on them. Moreover, in the case of autonomous vehicles, severe malfunctions almost inevitably lead to actual crashes [10]. We assume that node failures are detected at the transport/RPC layer, and that service calls have at-most-once semantics [11].

The mission controller may also fail during mission execution. From a traditional RPC perspective, this corresponds to a client failure, which is typically dealt with by garbage-collecting orphan calls at the server, using a suitable mechanism such as extermination or reincarnation [7]. In our case, when a node detects a failure of the mission controller, it enters a fail-safe state. Nodes remain in the fail-safe state until a human operator takes over control manually. However, it is desirable to tolerate failures of the mission controller in a transparent way, and continue the execution of the mission program without any human intervention. This would benefit a wide range of non-critical applications that can make extensive usage of autonomous vehicles, increasing the robustness and achieving a truly autonomic operation of such systems. The next section presents a solution to this problem.

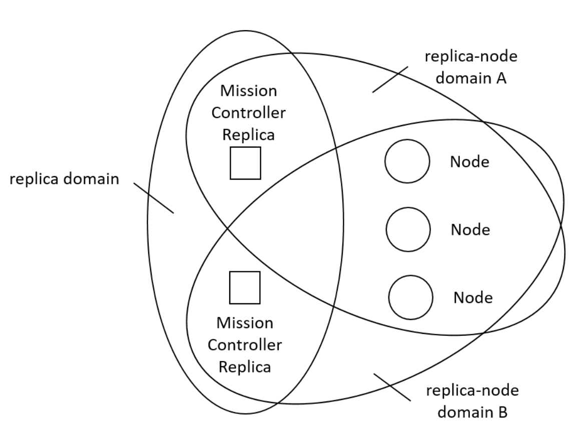

Fig. 1: Organization of the communication for the active replication of the mission controller.

III. Active Replication Approach

To tolerate failures of the mission controller, we propose an active replication approach where the mission program is actively executed by multiple instances of the mission controller. We refer to each instance of the mission controller as a *replica*. If one of the replicas fails, the remaining replicas can continue with the execution of the mission program.

Note that active replication is traditionally used in computer systems for servers/services that provide critical functionality [5]. In contrast, in our case the most critical system component is the mission controller, which according to the client-server model semantics acts as a client by invoking the services provided by the nodes.

A. Extensions to the basic system model

Figure 1 shows the organization of the communication that takes place between the mission controller replicas and the nodes, as this is assumed in our approach. Each replica interacts with the nodes within a separate/private communication domain. We refer to this as a *replica-node* domain. From a network layer perspective, a replica-node domain can be implemented as a VPN over the Internet, or via dedicated telecommunication links. In any case, given that nodes are mobile, at some point this unavoidably involves wireless communication. To avoid cross-talk between the domains and to increase reliability against jamming attacks, each domain could use a different radio frequency, possibly even a different radio technology for increased reliability.

Replicas interact with each other using a separate communication domain, the *replica* domain. Unlike the replica-node domains, the replica domain could be implemented using tethered-only networking technology (if the replicas are stationary). At the application layer, we assume standard reliable group communication support, such as reliable multicast [12]. The proposed mechanism (discussed in the sequel) works with simple FIFO delivery, without requiring more advanced/costly causal or total ordering [13].

We assume that reliable group communication comes with a reliable failure detection mechanism, ensuring that if a process (replica) fails then the failure will be announced to all other working processes (replicas) after the last message that was sent by the failed replica is delivered. In other words, for the replicas, we assume fail-stop failures with notification (we do not address byzantine failures of the mission controller).

B. Replication properties

In the traditional active replication scheme, several properties have been defined to capture the consistency requirements for a deterministic replicated service [14]. In our case, where the replicated entity is the mission controller, we capture the desired consistency and functionality via two properties:

Uniform Request Agreement. All working replicas issue the same requests in the same order.

Uniform Reply Integrity. All replicas that issue the same request, will receive the same reply.

Next, we present a mechanism that ensures these properties, for deterministic mission program executions. We start by considering executions without node failures, and then discuss the extensions that are needed to deal with node failures. Finally, we briefly discuss how to support the execution of non-deterministic mission programs.

C. Concurrent execution – without node failures

Assuming deterministic mission program execution, the replicas of the mission controller can execute the mission program in parallel to each other, without any synchronization (as long as there are no node failures, which are discussed in the sequel). This has the advantage of reduced communication overhead and faster mission progress. Due to this decoupled parallel execution, the execution of the mission program at some replicas may lag behind other replicas. We refer to a replica as *fast* if it issues a service call that has not yet been issued by another replica. A replica is called *slow* if it issues a call that has already been issued by some replica(s).

Let replica r_k keep a sequence number seq_k that is increased each time it performs a service call to a node. Replica r_f is fast if $seq_f \geq seq_k, \forall k$, whereas r_s is slow if $seq_s \leq seq_k, \forall k$. Of course, all replicas might issue requests at the same speed, in which case they are all equally fast/slow.

D. Duplicate service calls

Since the mission is deterministic, a node receives the same service request multiple times, once from each replica. However, each service call should be executed at most once.

To deal with such duplicate service calls, each node keeps a log of the requests received and the replies that were produced in return. Let $log[pos].seq$ and $log[pos].req$ be the sequence number and respectively a hash of the request at position pos in the log, and let $log[pos].rpl$ be the corresponding reply. Also, let pos_k be the log position for the last request of r_k. Finally, let pos_last be the position of the last log entry, which corresponds to the last new (not duplicate) request received from any replica.

When a node receives the next request req_k from r_k, it checks whether $pos_k = pos\_last$. If so, this is a new request, thus the respective service call is executed, the log pointer of r_k is incremented $pos_k = pos_k + 1$, and the request is appended to the log together with the reply that is sent back. If $pos_k < pos\_last$, the request of r_k has already been issued by another replica, and it is checked whether it is identical to the one stored in the next position of the log, $req_k = log[pos_k+1].req$. If so, the log pointer is incremented, $pos_k = pos_k+1$, and the corresponding reply $log[pos_k].rpl$ is retrieved from the log and is sent to the replica. Else, the node sends a reply indicating that this is an unexpected request, so that the mission controller can act accordingly. This can only occur in an exceptional scenario, which is discussed in more detail in Section III-F.

E. Node failures

When a node fails, it is no longer guaranteed that all replicas will continue their execution in the same way, even if the mission program is deterministic. This is because slow replicas will not be able to receive any replies from the failed node, and thus will notify the mission program about the node failure at an earlier point of execution compared to a faster replica. As a result, the mission program may take a different execution path than the one that was followed by the faster replicas, which may have successfully invoked the node before it failed. To address the problem, replicas have to synchronize to ensure that they will all continue the execution of the mission program in the same way. Note that the node failure may be discovered by a fast or a slow replica.

To support the required synchronization, every replica r_k also maintains a log with entries for the service calls it has performed to the nodes. Let $log[pos].seq$, $log[pos].n$ and $log[pos].rpl$ denote the call sequence number, the target node, and the reply that was received from the node, respectively. Also, every node n_i records the current call sequence number of each replica, and updates the minimum value, which is included in the replies sent to the replicas. Based on this information, in turn, each replica r_k maintains a conservative lower bound for the call sequence number of the slowest among all replica(s), let this be seq_slow.

When replica r_k detects that node n_f has failed, it stops the execution of the mission program, and enters a special synchronization state. After recording the failure, it sends to all replicas, via reliable multicast, a synchronization message that includes n_f and seq_k. The message also includes the log entries for all the calls r_k has issued to n_f that may have not yet been performed by some slower replica(s), i.e., all entries where $log[pos].n = n_f \wedge seq\_slow < log[pos].seq \leq seq_k$.

When a replica receives a synchronization message for node n_f for which it has not yet detected the failure, it acts as if it had just detected the failure of n_f (as discussed above). Also, for every synchronization message received, replicas add to their log any missing entries for the failed node n_f, update the sequence number of the last call that was issued by any replica to n_f, let this be $seq\_last[f]$, and update the sequence number of the slowest replica seq_slow.

The size of the synchronization messages exchanged between the replicas can be reduced using a simple optimization. Namely, if a replica learns about a node failure from another replica r_k, it suffices to include in its own synchronization message only the log entries for the failed node that are not already found in the message of r_k (which will be received by all replicas, thanks to the reliable multicast functionality).

Note that it is straightforward to handle the case where different replicas concurrently discover the failure of different nodes. In this case, all replicas remain in the synchronization state and repeat the process for the additional failed node(s) before resuming the mission program. Replica r_k remains in the synchronization state until it receives a synchronization message from every other replica. Then, it reverts to the normal state, and resumes the execution of the mission program.

Finally, the process of performing a service call is extended as follows. When the mission program issues a service call, before sending a request to the target node n_i, the replica checks whether it has recorded a failure of n_i. If not, the node is invoked as usual. Else, it is checked whether the log contains the reply for this request ($seq \leq seq\_last[i]$), in which case the reply $log[pos].rpl | log[pos].seq = seq$ is fetched from the log and is returned to the mission program. If the log does not contain the reply, an error is returned to the mission program, indicating that n_i has failed (the call sequence number is not incremented in this case).

F. Replica failures

When a node detects the failure of a replica r_f, it simply removes r_f from the set of replicas from which it expects to receive requests. It may also garbage collect all internal data structures concerning r_f. Nodes will enter the fail-safe state only when all replicas of the mission controller fail.

When a replica detects the failure of another replica r_f, it removes r_f from the set of working replicas in order for this replica to be excluded from subsequent communication/synchronization rounds. Note that if a replica fails during a synchronization round, thanks to the reliable multicast functionality, its message will either be received by all working replicas or by none of them, so this will not affect the outcome.

However, there is a corner case where the failure of a replica will cause a problem, namely if node n_f fails, and there is a single fast replica r_f, which is the only one that has performed the most recent service call(s) to n_f, and r_f fails too, before it manages to send its own synchronization message with the missing log entries for n_f. In this case, the remaining replicas cannot recover these entries and there is no safe way for them to deduce that this is a problematic situation. Thus, they continue with the execution of the mission program as usual. However, this may lead to a different execution path from the one followed by the fast replica r_f. This situation is detected when a replica receives a reply indicating that its request is unexpected (see Section III-D). Then, the mission program is notified in order to handle the problem (e.g., set the nodes in fail-safe mode, and retrieve information from them to assess the current situation). If the prospect of such a discontinuity is not acceptable, one has to adopt the more conservative approach for supporting non-deterministic execution, discussed in Section III-H.

G. Garbage collection of log entries

The logs of the nodes and the replicas cannot grow indefinitely. Fortunately, both logs can be garbage-collected in a straightforward way, without any additional communication.

Nodes can remove a log entry once a corresponding service call request is received from every working replica. More specifically, all entries with $log[pos].seq \leq seq\_min$ can be safely removed from the log. This way, the log of a node only contains entries for the service calls that have not yet been performed by the slowest replicas.

Along the same lines, a replica can truncate its log up to the entry for which $log[pos].seq = seq\_slow$, which represents a conservative lower bound for the last call that has been performed by the slower replicas. Recall that seq_slow is calculated independently by each replica, based on the information that nodes append to their replies. Also note that during the synchronization phase, all replicas can update seq_slow to accurately reflect the smallest call sequence number among all replicas, and thus can safely truncate their log accordingly.

H. Non-deterministic execution

The above approach will only work if the mission program is deterministic. But requiring the mission logic to be deterministic can be restrictive. Moreover, as discussed above, there is a corner case that may lead to mission discontinuation.

Non-deterministic execution can be supported by adopting a semi-active replication approach [15]. Adapted to our context, this works as follows. When a non-deterministic operation is encountered in the mission program, the replicas pause their execution in a barrier-like manner. Then, a distinguished replica, the *leader*, executes the non-deterministic part of the mission program, and when done transfers the local execution state to the *follower* replicas (again, this can be done using simple reliable FIFO multicast). The followers, in turn, update their state and resume the execution of the mission after the endpoint of the non-deterministic execution.

If the leader fails, a new one is elected. Given that any replica can assume this role, the election can be done based on trivial information that is already available locally at each replica. For example, the role of the leader can be assigned to the replica with the largest identifier.

The nodes keep the same log structure as discussed above in order to avoid executing duplicate service calls during the execution of deterministic sections. In addition, the requests issued by the leader during a non-deterministic section are flagged so that the next request of a follower replica, when it resumes execution after a non-deterministic section, will not be detected as unexpected (there will be a gap in the sequence numbers of the previous and the next request of the replica, before and respectively after the non-deterministic section). Note that if the leader fails while executing a non-deterministic section, the new leader that takes over will perform the last service call that may have already been performed by the previous leader before it failed. Such duplicates can be detected and handled by the node as usual.

As will be shown in Section V, this mode of operation comes at a significant cost. Therefore, it is important for missions to be designed so that they have few, well-defined non-deterministic sections. The worst case is for the entire mission program to be non-deterministic, in which case the synchronization between the leader and the followers has to be performed at every service call towards a node.

IV. Implementation

We have implemented the proposed replication approach in a suitably extended version of the TeCoLa programming framework [16]. TeCoLa is a Python-based middleware, designed to facilitate the high-level coordination of dynamic and heterogeneous robotic teams.

For the interaction between the mission controller and the nodes we employ GCBRR, a reliable 1-N request/reply transport with group management capabilities [17]. GCBRR is designed for channels with physical multicast capability, and supports 1-to-N request/reply exchanges with the minimum number of message transmissions, at low latency and without any contention among the communicating parties. Each service call performed by the mission programs is mapped, behind the scenes, to a corresponding request/reply interaction.

The replicas of the mission controller are specified in a configuration file (the current implementation does not support the dynamic addition of replicas at runtime). Reliable multicast communication between the replicas of the mission controller is implemented using the JGroups toolkit [18]. We configure JGroups to use UDP/IP as transport to exploit IP multicasting, and to employ the NAKACK2 protocol for the reliable FIFO delivery of multicast messages using negative acks.

To support non-deterministic mission programs, a replica must be able to save and restore the state of the mission program. In our implementation, this is done using the the DMTCP framework [19]. To accelerate prototyping, we save the entire state of the process that runs the mission program in a brute-force way, without attempting a more elaborate integration with the TeCoLa environment (which might allow a more selective recording of the absolutely crucial state information). To reduce the size of the images, we perform incremental checkpointing using the HBICT module [20], which works seamlessly with DMTCP.

V. Cost Analysis

In this section we discuss the communication overhead of the proposed active replication approach. On the one hand, we give analytical estimations for the main components of the mechanisms. On the other hand, we record the overhead of the current implementation in TeCoLa, and compare them with the analytical estimates. Our analysis assumes that a node handles incoming requests in a serialized/FIFO manner, thus, the handling of the next request starts after the handling of the previous request is completed (as done in the implementation). Also, we assume that the replica-node domains are isolated and do not cross-talk/interfere with each other.

A. Experimental setup

The experimental measurements are performed using the AeroLoop simulation environment [21]. AeroLoop allows the developer to test mission software by running experiments with several virtual unmanned aerial vehicles (vUAVs) that can be controlled from a virtual ground station (vGS). In our case, the vUAVs and the vGS run the TeCoLa software stack.

We introduce multiple vGS, each running a replica of the TeCoLa mission controller. The replica-node domains are implemented as separate wireless WiFi networks, with the simulation support of NS3. The WiFi rate is set at the basic rate for multicasts, 1 Mbps. The replica domain is implemented as a wired Ethernet network with a rate of 210 Mbps.

The mission program that runs on top of TeCoLa, performs a series of dummy service calls.We set the requests and replies either so that their payload fully occupies the maximum packet payload, or so that their payload is empty, in which case the respective packets only carry the basic protocol headers. We can also set the amount of processing that is performed by the node for each such call. In our experiments, we vary the number of replicas used for the mission controller. Note that the overhead of the proposed replication scheme does not depend on the number of nodes employed in the mission.

B. Service call delay in deterministic execution

First, we investigate the end-to-end delay of a service call for deterministic execution scenarios, for the typical case where the target node is alive. This can be expressed as $T_{call} = T_{rtt} + T_{proc}$. The round-trip-time T_{rtt} is the time it takes to send the request and receive the reply over the communication channel of the node-replica domain. For a fast replica, T_{proc} equals the time needed by the node to process the request. For a slow replica, $T_{proc} = 0$ if the node is idle, as the reply is directly fetched from the log. If, however, the node is busy processing a request of a fast replica, in the worst case, T_{proc} will be equal to the respective processing delay.

In a first set of experiments, we measure T_{call} for the case where there are no node failures. We use three different service calls, which perform a brute-force primality test, with a processing time T_{proc} of 1, 2 and 3 seconds, respectively. Each request and reply carries the maximum packet payload for WiFi, 1500 bytes, yielding a T_{rtt} of 26 milliseconds at the WiFi channel rate of 1 Mbps. We employ two replicas of the

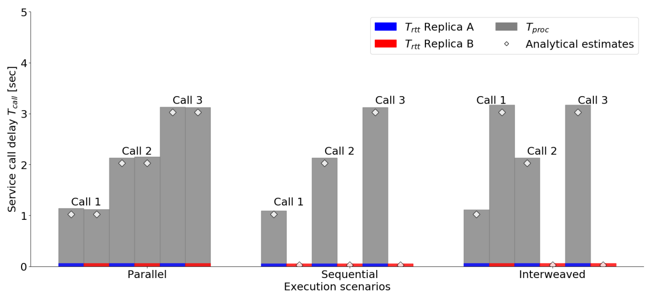

Fig. 2: Service call delay in the absence of node failures.

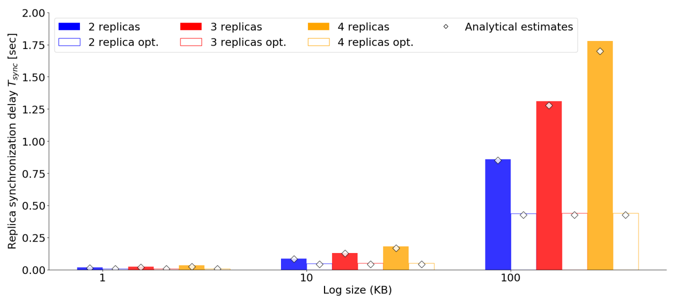

Fig. 3: Replica synchronization delay on node failure.

mission controller (A and B), running a mission program that performs these service calls, one after the other. We experiment with three execution scenarios. In the *parallel* execution, both replicas perform the same call practically at the same time. In the *sequential* execution, replica A performs the first service call, and once this returns then replica B proceeds to perform the same call. Replica A performs the next service call right after the completion of the previous call at replica B. In the *interweaved* execution, replica A is two calls ahead of replica B, and replica B performs the first service call when replica A already performs the last call.

Figure 2 reports the analytical and experimental results. In the parallel execution scenario, the call delays at both replicas are the same, and equal to the call delay when using a single replica. In the sequential execution, replica B experiences significantly lower service call delays, which basically amounts to the round-trip time. This is because these calls have already been processed by the node due to the calls performed by replica A, so the node simply returns the replies from the log. Finally, in the interweaved execution, for the first call, the slower replica B experiences the same delay as the faster replica A for the third (more time-consuming) call, while the rest of the calls execute very fast, like in the sequential execution scenario.

It is important to stress that the experimentally measured delays are close to the ones estimated analytically. In general, the service call delay is strictly bounded by the processing time of the most time-consuming service call. Also note that any additional replica(s) would experience a call delay within the lower and upper bounds reported here, depending on the time of invocation with respect to the faster replica.

C. Replica synchronization delay for node failures

Once a node failure has been detected and the replica synchronization has been completed, all subsequent calls of the mission program to that node are handled based on the local log of the mission controller, without any communication, thus T_{call} is negligible. Therefore, we focus on the overhead of the synchronization between the replicas.

We analytically estimate $T_{sync} = N \times T_{log}$, where N is the number of replicas and T_{log} is the time it takes for a replica to send its own log entries to all other replicas via reliable multicast. This, in turn, can be expressed as $T_{log} = LogS/MaxP \times T_{rm}$, where $LogS$ is the total size of the log entries to send, $MaxP$ is the maximum packet payload for the underlying network, and T_{rm} is the time it takes to send a reliable multicast message that occupies a full packet. With JGroups configured to exploit IP multicasting and negative acknowledgements, each reliable multicast roughly translates to a single packet transmission, yielding a T_{rm} of about 6 milliseconds for a fully loaded packet of 1500 bytes over the 210 Mbps Ethernet network of the replica domain.

In a second series of experiments, we measure T_{sync} for 2, 3 and 4 replicas running a mission program that periodically invokes a node. We also vary the total size of the log entries that need to be exchanged between the replicas ($LogS$), from 1 KB, 10 KB up to 100 KB. Note that $LogS$ depends on the number of service calls performed by the fastest replica to the node (before it failed) which have *not* yet been performed by the slowest replica, as well as on the size of the node's replies to these calls. But what actually matters is the total size of this log information.

Figure 3 shows the results (average over 20 runs; there is no significant deviation) for the naive and optimized version of the synchronization protocol (see Section III-E). The analytical estimates are, as before, very close to the measured delays. We observe that the synchronization delay increases linearly to the size of the replica logs and the number of replicas. Notably, the optimized version reduces the synchronization delay significantly, which practically becomes constant irrespectively of the number of replicas —for every additional replica the increase is lower than 0.01%. This becomes clearly visible for larger log sizes. The reason is that only the replica that detects the failure first, includes in its synchronization message the log entries for the failed node, while all other replicas do not re-send the same entries and thus generate very small synchronization messages.

D. Service call delay in non-deterministic execution

The delay of a service call in the non-deterministic execution mode can be expressed as $T_{call} = T_{checkpoint} + T_{rtt} + T_{proc}$. As above, $T_{rtt} + T_{proc}$ represents the time it takes to perform the actual call to the node. In addition, one has to pay the cost of a checkpoint operation $T_{checkpoint}$, which can be expressed as $T_{rec} + T_{transfer}$, where T_{rec} is the time it takes for the leader replica to record its state, and $T_{transfer}$ is the

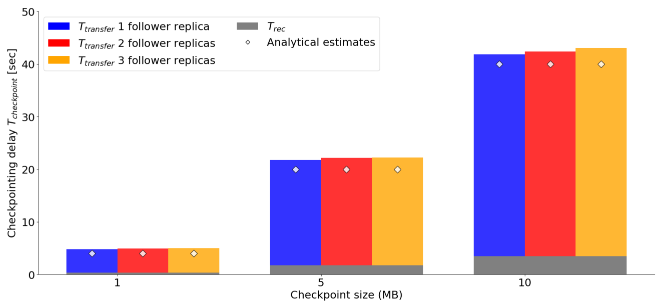

Fig. 4: Checkpointing delay in non-deterministic execution.

time needed to transfer the checkpoint image to the follower replicas via reliable multicast. The state recording delay T_{rec} depends on the number and size of data objects that were created/modified by the mission program. The image transfer delay can be expressed as $T_{transfer} = ImageS/MaxP \times T_{rm}$, where $ImageS$ is the size of the image, $MaxP$ is the maximum packet payload for the underlying network, and T_{rm} is (as above) the time it takes to send a single reliable multicast message that occupies a full network packet.

We have measured T_{call} for a null service, with $T_{proc} = 0$ and $T_{rtt} = 8$ milliseconds for an empty service call request and reply over the 1 Mbps WiFi network. We do this for 2, 3 and 4 replicas (1, 2 and 3 follower replicas respectively). Also, we artificially vary the size of checkpoint images, from 1 MB, 5 MB up to 10 MB, which are representative sizes for several test applications we have programmed in TeCoLa.

Figure 4 reports the results together with the analytical estimates. Given that the cost for performing the call itself is negligible compared to the checkpoint delay $T_{checkpoint}$, we only show the latter, broken down to the state recording delay T_{rec} and the image transfer delay $T_{transfer}$. As expected, the delay grows linearly to the size of the checkpoint image. Note that the number of replicas do not affect the checkpointing delay significantly; for every additional replica, the increase is lower than 1% thanks to the efficient underlying reliable multicast implementation (as discussed above, $T_{rm} = 6$ milliseconds). However, it is clear that taking a checkpoint at every service call incurs a significant penalty, especially when the execution state is large. It is thus important for the mission program to accurately indicate the parts that are non-deterministic, allowing the system to adopt the mode of deterministic execution as much as possible.

VI. Related Work

Extensive research has been done in regard to fault tolerance in distributed systems. Among the most well studied techniques are rollback recovery [22] and replication [3]. Rollback protocols assume a stable storage that is used to store recovery information during normal execution. After a failure occurs, the failed component uses this information to restart its execution from a more recent state. To reduce the extent of rollback and guarantee that the pre-failure execution can be deterministically regenerated, log-based protocols [23] have been proposed, which combine checkpointing with logging and replaying of non-deterministic events.

Software-based replication achieves fault-tolerance of critical components by employing multiple instances that can fail independently. In passive replication [4], one of the replicas, called the primary, is responsible for receiving and processing input and producing output. The remaining, called the backups, are merely notified to apply the changes produced by that processing. On the contrary, active replication [5] is a non-centralized approach where all the replicas receive and process (concurrently) the same sequence of inputs. This leads to masking the failures and achieving better performance, making it ideal for (soft) real-time applications. However, it requires that the output produced by all the replicas is the same, i.e. the processing is deterministic. In addition, there are variations that combine elements from both replication strategies [15].

While the vast majority of the replication-related bibliography focuses on replicating the server side, in our work we apply replication to the client side (the mission controller), which is the most critical component. Also, traditional active replication requires coordination between the replicas during each request in order for them to be handled in the same order, whereas in our approach the replicas need to synchronize only when a node fails. Finally, in order to support non-deterministic applications, we adapt our approach to a semi-active technique supported by appropriate checkpointing.

There are recent research efforts on making byzantine fault-tolerant (BFT) systems practical, by improving their performance [24], robustness [25], and resource efficiency [26]. However, fail-stop failures are a realistic assumption for our work, given that the embedded/cyber-physical systems we discuss usually have built-in mechanisms for locally detecting faults, failures or even malicious attacks. For instance [8] proposes a model-based estimation approach for detecting misbehaviors/malfunctions of sensor and actuators in mobile robots, while [27] presents a framework to achieve detection of both software and hardware failures, and even achieve fault mitigation through self-adaption or cooperation between multiple robots. Notably, in UVs, severe malfunctions almost inevitably lead to actual crashes [10].

In swarm robotics, tolerance to faults of single robots is to a large degree built-in. Failures are usually detected through various, often biologically inspired, techniques [28], [29], which exploit the natural redundancy of the (large) swarm. Since no predefined roles are assigned to the robots, reorganization is achieved in a self-healing fashion [30], [31].

There are also works targeting multi-robot collaborative systems. ALLIANCE [32] is a software architecture that facilitates cooperative control of teams of mobile robots for achieving fault tolerance. It is a distributed, behavior-based architecture that allows each robot to adapt its actions during a mission. This way, if a robot fails, its tasks are dynamically re-allocated to the remaining team members. [33] presents a programming abstraction for handling failures in ensembles of robots. The proposed abstraction allows the application programmer to annotate code blocks that include critical actions and define compensating actions in case a failure occurs. However, in both approaches, the fault handling and recovery is the responsibility of the developer, whereas our work practically achieves full transparency for the developer of the mission program.

Finally, [34] presents a passive replication approach for centrally coordinated robotic systems and deterministic mission programs. A combination of checkpointing and logging is employed to support the replay of the mission program in case of a rollback due to a failure of the mission controller. Here, we follow an active replication strategy, and additionally support non-deterministic mission programs.

VII. Conclusion

Our work focuses on systems that employ autonomous unmanned vehicles (UVs) which are centrally coordinated by a computer-based mission controller, and proposes an active replication scheme for tolerating failures of the mission controller, for both deterministic and non-deterministic executions. We also discuss an implementation of the proposed approach and identify the main overheads, analytically and through simulation experiments. For deterministic execution, as long as no UV fails, our mechanism does not introduce any overhead to the execution of the mission compared to using a single mission controller. When a UV fails, the synchronization overhead mainly depends on the size of the log entries that need to be sent from the fastest replica of the mission controller to the slower ones. For non-deterministic execution, the overhead is practically dominated by the time it takes to transfer the execution state of the mission controller to other replicas.

Since the overhead in deterministic and non-deterministic execution depends on the size of the logs and checkpoint images, respectively, we wish to investigate ways to reduce these sizes. We also wish to evaluate our approach for different radio technologies, in particular for the replica-node communication domains. Last but not least, we plan to test and evaluate our implementation in the real world, using an appropriate testbed.

References

[1] https://www.sensefly.com/industries/case-studies.

[2] E. Şahin, "Swarm robotics: From sources of inspiration to domains of application," in *Swarm Robotics*, 2005, pp. 10–20.

[3] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software Technologies — Ada-Europe '96*, 1996, pp. 38–57.

[4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd ed.)," 1993, ch. The Primary-backup Approach.

[5] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[6] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[7] B. Nelson, "Remote procedure call," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1981.

[8] P. Guo, H. Kim, N. Virani, J. Xu, M. Zhu, and P. Liu, "RoboADS: Anomaly detection against sensor and actuator misbehaviors in mobile robots," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

[9] M. L. Fairbairn, I. Bate, and J. A. Stankovic, "Improving the dependability of sensornets," in *IEEE International Conference on Distributed Computing in Sensor Systems*, 2013.

[10] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Conference on Security Symposium*, 2015.

[11] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 381–404, 1983.

[12] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 251–273, 1984.

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[14] X. Défago and A. Schiper, "Semi-passive replication and lazy consensus," *Journal of Parallel and Distributed Computing*, vol. 64, no. 12, pp. 1380–1398, 2004.

[15] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *International Conference on Distributed Computing Systems*, 2000.

[16] M. Koutsoubelias and S. Lalis, "Tecola: A programming framework for dynamic and heterogeneous robotic teams," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2016.

[17] ——, "Coordinated broadcast-based request-reply and group management for tightly-coupled wireless system," in *International Conference on Parallel and Distributed Systems*, 2016.

[18] http://jgroups.org.

[19] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *International Symposium on Parallel Distributed Processing*, 2009.

[20] http://hbict.sourceforge.net.

[21] M. Koutsoubelias, N. Grigoropoulos, and S. Lalis, "A modular simulation environment for multiple UAVs with virtual WiFi and sensing capability," in *Sensors Applications Symposium*, 2018.

[22] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[23] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, pp. 7:1–7:39, 2010.

[25] P.-L. Aublin, S. B. Mokhtar, and V. Quema, "RBFT: Redundant byzantine fault tolerance," in *IEEE International Conference on Distributed Computing Systems*, 2013.

[26] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient byzantine fault tolerance," in *ACM European Conference on Computer Systems*, 2012.

[27] Y. Cui, R. M. Voyles, J. T. Lane, and M. H. Mahoor, "ReFrESH: A self-adaptation framework to support fault tolerance in field mobile robots," in *International Conference on Intelligent Robots and Systems*, 2014.

[28] A. Christensen, R. O'Grady, and M. Dorigo, "From fireflies to fault-tolerant swarms of robots," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 754–766, 2009.

[29] A. Khadidos, R. M. Crowder, and P. H. Chappell, "Exogenous fault detection and recovery for swarm robotics," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 2405–2410, 2015.

[30] J. D. Bjerknes and A. F. T. Winfield, "On fault tolerance and scalability of swarm robotic systems," in *Springer Tracts in Advanced Robotics*, 2013, pp. 431–444.

[31] J. Timmis, A. Ismail, J. Bjerknes, and A. Winfield, "An immune-inspired swarm aggregation algorithm for self-healing swarm robotic systems," *Biosystems*, vol. 146, pp. 60–76, 2016.

[32] L. E. Parker, "Alliance: an architecture for fault tolerant multirobot cooperation," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 2, pp. 220–240, 1998.

[33] N. Beckman and J. Aldrich, "A programming model for failure-prone, collaborative robots," in *International Workshop on Software Development and Integration in Robotics*, 2007.

[34] M. Koutsoubelias and S. Lalis, "Fault-Tolerance Support for Mobile Robotic Applications," in *International Symposium on Industrial Embedded Systems*, 2018.

Appendix A
Algorithmic description

Below, we give an algorithmic description of the active replication mechanism for deterministic mission programs.

Algorithm 1 shows how a node handles a service call request coming from a replica of the mission controller, while Algorithm 2 shows how a replica handles the reply that is sent from a node in response to a previously issued request.

Algorithm 1 Handling incoming service calls at the node

```
1: $log \leftarrow \emptyset$                                ▷ request/reply log
2: $pos\_last \leftarrow 0$                                   ▷ last log position
3: $seq\_min \leftarrow 0$            ▷ smallest sequence number over all replicas
4: for each replica $r_k$ do
5:     $pos_k \leftarrow 0$                         ▷ position of last request of $r_k$
6: end for

7: upon receiving $\langle REQUEST, seq_k, req_k \rangle$ from $r_k$ do
8:     if $pos_k = pos\_last$ then                              ▷ new request
9:         $rpl_k \leftarrow$ execute $(req_k)$
10:        $pos\_last \leftarrow pos_k$
11:        $pos_k \leftarrow pos_k + 1$
12:        append$(log, (seq_k, req_k, rpl_k))$
13:        $seq\_min \leftarrow$ min$(log[pos_k].seq | \forall k)$
14:        send $\langle REPLY, seq_k, rpl_k, seq\_min \rangle$ to $r_k$
15:    else if $pos_k < pos\_last$ then                         ▷ old request
16:        if $req_k = log[pos_k + 1].req$ then
17:            $pos_k \leftarrow pos_k + 1$
18:            $seq\_min \leftarrow$ min$(log[pos_k].seq | \forall k)$
19:            send $\langle REPLY, seq_k, log[pos_k].rpl, seq\_min \rangle$ to $r_k$
20:        else                              ▷ mismatch with logged request
21:            send $\langle UNEXPECTED\_REQ, seq_k \rangle$ to $r_k$
22:        end if
23:    end if
24: end
```

Algorithm 2 Handling node replies at the replica

```
1: $log \leftarrow \emptyset$
2: $seq\_slow \leftarrow 0$             ▷ sequence number of slowest replica(s)

3: upon receiving $\langle REPLY, seq, rpl, seq\_min \rangle$ from $n_i$ do
4:     append $(log, (seq, n_i, rpl))$
5:     $seq\_slow \leftarrow$ max$(seq\_slow, seq\_min)$
6:     return-to-mission-program $rpl$
7: end

8: upon receiving $\langle UNEXPECTED\_REQ, seq \rangle$ from $n_i$ do
9:     return $UnexpectedReqError$            ▷ notify problematic situation
10: end
```

Algorithm 3 shows the synchronization between replicas when a node failure is detected. This includes the optimization that avoids sending superfluous log entries when a replica learns about a node failure as a side-effect of receiving the synchronization message of a faster or equally fast replica. The code also handles the case where different node failures are detected simultaneously. Then, multiple synchronization rounds are performed in parallel, one round for each node. A replica returns to normal mode when all rounds complete, i.e., once it receives a synchronization message from every working replica for each failed node.

Algorithm 3 Replica synchronization for a node failure

```
1: $state \leftarrow normal$
2: for each node $n_i$ do
3:     $failed[i] \leftarrow false$               ▷ own failure detection flag for $n_i$
4:     $sync[i] \leftarrow 0$     ▷ nof sync messages received for the failure of $n_i$
5:     $seq\_last[i] \leftarrow 0$   ▷ sequence number of last call to $n_i$ in the log
6: end for

7: upon detecting failure of node $n_f$ do
8:     if $failed[f] = false$ then
9:         $state \leftarrow sync$
10:        $failed[f] \leftarrow true$
11:        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
12:        $log_f \leftarrow$ getLogEntries$(log, n_f, seq\_slow, seq)$
13:        send $\langle SYNC, n_f, seq, log_f \rangle$ via RM
14:        if $(failed[i] = false) \vee (sync[i] =$ nofReplicas$()), \forall n_i$ then
15:            $state \leftarrow normal$
16:        end if
17:    end if
18: end

19: upon receiving $\langle SYNC, n_f, seq_k, log_k \rangle$ from $r_k$ do
20:    $sync[f] \leftarrow sync[f] + 1$
21:    $seq\_slow = min(seq\_slow, seq_k)$
22:    if $seq < seq_k$ then
23:        appendLogEntries$(log, log_k, seq, seq_k)$
24:        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
25:    end if
26:    if $failed[f] = false$ then
27:        $state \leftarrow sync$
28:        $failed[f] \leftarrow true$
29:        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
30:        if $seq > seq_k$ then
31:            $log_f \leftarrow$ getLogEntries$(log, n_f, seq_k, seq)$
32:        else
33:            $log_f \leftarrow \emptyset$
34:        end if
35:        send $\langle SYNC, n_f, seq, log_f \rangle$ via RM
36:    end if
37:    if $(failed[i] = false) \vee (sync[i] =$ nofReplicas$()), \forall n_i$ then
38:        $state \leftarrow normal$
39:    end if
40: end
```

Finally, algorithm 4 shows the extended service call handling at the replicas that exploits the log to retrieve the replies of failed nodes (which were added to the local log via the above synchronization). If the mission program invokes a failed node and the log does not contain the reply for that call, an error is returned indicating the node failure (handled by the mission program as usual).

Algorithm 4 Extended service call process at the replica

```
1: when invoking $n_i$ with request $req$ do
2:     $seq \leftarrow seq + 1$
3:     if $failed[i] = false$ then                   ▷ issue request as usual
4:         send $\langle REQUEST, seq, req \rangle$ to $n_i$
5:     else if $seq \leq seq\_last[i]$ then          ▷ get reply from the log
6:         return-to-mission-program getLogReply$(log, seq)$
7:     else
8:         $seq \leftarrow seq - 1$
9:         return $NodeFailureError$                  ▷ indicate node failure
10:    end if
11: end
```