Heartbeat Bully: Failure Detection and Redundancy Role Selection for Network-Centric Controller

Bjarne Johansson[1,2], Mats Rågberger[1], Alessandro V. Papadopoulos[2], Thomas Nolte[2]
[1] ABB Industrial Automation, Process Control Platform, Västerås, Sweden
[2] Mälardalen University, Västerås, Sweden
{bjarne.johansson,mats.ragberger}@se.abb.com,
{alessandro.papadopoulos, thomas.nolte}@mdh.se

***Abstract*—High availability and reliability are fundamental for distributed control systems in the automation industry. Redundancy solutions, with duplicated hardware, is the common way to increase availability. With the advent of Industry 4.0, the automation industry is undergoing a paradigm shift; a peer-to-peer mesh oriented architecture is replacing the traditional hierarchical automation pyramid. With generic computational power provided anywhere in the cloud — device continuum, the conventional control centric solutions are becoming obsolete. The paradigm shift imposes new challenges and possibilities on the redundancy solutions used. We present and evaluate a hardware-agnostic algorithm suitable for failure detection and redundancy role selection in the new automation paradigm. The algorithm is modeled, evaluated and validated with the model checking tool UPPAAL.**

I. INTRODUCTION

Distributed Control Systems (DCSs) are often part of the automation solution in domains where unplanned downtime is costly, e.g., oil extraction and petroleum production. A common way to increase availability is critical hardware multiplication, for example, controller duplication. In practice in the DCS domain, the most commonly used redundancy scheme is the one out of two (1oo2) pattern. The 1oo2 pattern is a specialization of the M out of N (MooN) pattern, where N nodes are in a passive mode, ready to take over for the M active nodes. In a redundant DCS system, the active primary controller continuously synchronizes with the backup for fast take over in case of primary failure. The synchronization function typically utilizes a purpose-fit redundancy link.

When the automation industry and DCS transcend into the age of Industry 4.0 [1] and Industrial Internet [2], the wake that follows brings the possibility of more elastic computational power with edge [3], fog [4] and cloud computing. The increased information and data exchange that are essential concepts in Industry 4.0 and Industrial Internet is boosting the incitement for replacing the traditional controller-centric architecture with a network-centric architecture. Using purpose-fit, dedicated, and highly customized communication links for redundancy purposes impose hardware requirements and by that deployment limitations.

Replacing dedicated redundancy link hardware with general-purpose communication links such as Ethernet and hardware-agnostic software reduces the hardware requirement and increases the deployment alternatives. However, doing so means that the dynamic state transfer from primary to backup, failure detection, and role selection, functionality previously provided by the links becomes software responsibility. State transfer between the redundant process is a topic of its own and not discussed further in this paper. For the remainder of the paper, the term *process* denotes the execution context hosting the control engine. The providing of computational power and messaging capabilities to the processes is outside of the scope.

Model checking is a method suitable for formal verification of algorithm properties. UPPAAL is a model checking tool for stochastic model checking, using timed automata [5], [6]. A timed automaton is a finite state machine with clocks and time constraints [7]. UPPAAL supports modeling of a network of timed automata, i.e., multiple timed automata that can synchronize over channels. With a network of timed automata, process interaction models easily, for example, a client-server protocol interaction. The support of network timed automata makes UPPAAL suitable for modeling our algorithm and the interaction between primary and backup processes.

Our contribution is an algorithm that utilizes the failure detection for role selection. In case of primary failure, it deterministically elects the new primary out of a plurality of backup processes within a known upper-bounded time. We start by formulating the problem as functionality needed and requirements on deterministic behavior. Then we describe the algorithm and provide an UPPAAL model. The UPPAAL model checking verifies the functionality and determinism requirements.

The outline of the paper is as follows. The overview of the related work is provided in Section II followed by a problem description in Section III. The proposed algorithm is described in Section IV. The UPPAAL model verification is described in Section V. Lastly, we provide a conclusion and potential future work in Section VI.

II. RELATED WORK

The redundancy role selection problem is very similar to the leader election problem, which is a well-known problem in distributed systems [8]. The goal of the leader election procedure is that all processes agree upon a new leader process

This work is funded by The Knowledge Foundation (KKS), project ARRAY, and by The Swedish Foundation for Strategic Research (SSF).

upon failure of the current leader, and ensure that all processes share the view of which process is the new leader. Translated to redundant controllers and the DCS domain, the leader is the primary. The goal is to select a new primary, amongst a plurality of backups, in case of primary failure.

There exist many leader election solutions. One of the first is the algorithm proposed by Robert and Chang [9], and another well known algorithm is the Bully algorithm presented by Garcia-Molina [10]. Both algorithms were introduced in the late 70s and early 80s, and much has happened since then. The Fast Bully Algorithm (FBA) presented by Lee et al. [11] is one example, and there are many other variants [12], [13], [14], [15], [16].

A common way to benchmark leader election algorithms is to measure the number of messages needed to elect a new leader. The Bully algorithm, in the worst case, requires n^2 messages to choose a new leader, and FBA needs n messages, with n being the number of leader candidate processes in the distributed system. In the DCS domain and redundancy, n would be the number of processes forming the redundant solution. However, for real-time functions, such as electing a new primary controller processes, time and deterministic behavior is more important than the number of messages.

The failure of the current leader must be detected to initiate an election, i.e., failure detection is needed. Typically a silent leader is assumed to have failed. Chandra et al. [17] present two different properties that are fundamental for failure detectors: completeness and accuracy.

Completeness is the failure detection degree, divided into strong and weak. When guaranteed that every non-faulty process permanently suspects every faulty process, the failure detection completeness is strong. Weak completeness failure detection is the degree reached when every faulty process will be detected by at least one non-faulty process.

Accuracy is divided into four levels, where false positive means that a non-faulty process is suspected of having failed:

- Strong. No indication is a false positive.
- Weak. Not all indications are false positives.
- Eventual strong. After a stabilization period, there are no false positives.
- Eventual weak. After a stabilization period, not all indications are false positives.

Earlier work prove the impossibility of having a strong completeness and strong accuracy in a asynchronous distributed system [18], [19]. A synchronous distributed system is a distributed system that fulfills the criteria below:

- Deterministic network. There is a known, upper bound, message delivery time.
- Ordered message delivery. An older message is never processed, by the receiver, after a newer.
- Global time. All processes know the current time.

Two main monitoring failure detection approaches exist, push and pull [20]. Push is when the supervised process publishes a message with an expected rate, often referred to as a heartbeat, that the supervising processes can monitor. An example of a pull approach is when the monitoring processes send a "how are you" message to the supervised process, if the answer is missing or erroneous, the monitored process is suspected to be in a non-healthy state.

There exist many variants of failure detection algorithms, ranging from algorithms for reducing network load with message piggybacking to statistics based strategies to reduce false positives on unreliable networks [20], [21], [22], [23], [24], [25], [26].

The control logic execution process in a DCS is a real-time process, i.e., the function output and the temporal aspect are equally important. The real-time aspect yields that the failure detection, besides from being reliable and deterministic, also needs to detect the failure within an upper bound time. Failure detection time adds to the total take over time, which is the time from a primary failure until a backup has resumed as the new primary, also referred to as failover time. During failover, the controller does not produce any output, i.e., the controlled process must tolerate frozen outputs during the failover. Our work bridges the previous failure detection and leader election work to our specific domain by proposing an algorithm tailored for the domain. The difference compared to the earlier work is that our algorithm utilizes the failure detection for the role selection and relies on the real-time capabilities that a control system needs to fulfill.

III. Problem formulation

We consider a set R of N processes forming a redundant configuration. $P \subseteq R$ denotes the set of primary processes and $B \subseteq R$ the set with backup processes. At all times, there should be at most one primary.

$$\begin{aligned} P \cup B &= R \\ P \cap B &= \emptyset \\ |P| &\leq 1 \end{aligned} \tag{1}$$

If the primary fails, a new primary should be selected in a deterministic way amongst a plurality of backups. A primary that fails is no longer considered a primary and is assumed to fail silently.

The algorithm needs to support the following.

- Failover time. Known upper bound time from primary failure to a new primary has been selected. Note that, in this paper, we only consider the time from the silence of the primary, until the backup that will become primary is informed.
- Failover determinism. The backup to primary role precedence should be known, i.e., the backup to become the new primary should not be arbitrary.
- Switchover. A primary should be able to pass on the primary role to a backup process. The switchover possibility is a common feature in today's redundant DCS for a controlled change of the primary.
- Clinging. The current primary should remain, even if a former primary recovers.
- Single primary. Only one primary at all times.

The algorithm rely on the following assumptions.

- Byzantine free. None of the processes are, intentionally nor unintentionally, malicious.
- Message ordering. Messages order is kept. If a process sends message M_i before message M_j, then the receiving process either receives message M_i before message M_j, or M_i is discarded. I.e., not processed by the receiving process at all. Message ordering is assumed to be the communication protocol's responsibility.
- Reliable communication. Redundant communication paths ensure that the probability for message loss is negligible. Message loss due to communication failure leads to partitioning, i.e., multiple sets of redundant processes that are unable to communicate in-between the sets. Message propagation time, including processing in sender and receiving side, is assumed negligible to the messaging cycle time.
- Accurate clocks. Synchronized time is not needed, but the perception of elapsed time needs to be equal enough for the drift to be negligible.

The above are properties of a reliable real-time system, and if fulfilled by a distributed system, the system is synchronous. If the system is synchronous, the failure detection can achieve strong accuracy and completeness [17]. We use those properties to propose a failure detection based role selection.

IV. Heartbeat bully

We named the algorithm *Heartbeat bully* since it utilizes heartbeat-based failure detection for redundancy role selection, inspired by the Bully leader election algorithm [10]. Similar to the Bully algorithm, *Heartbeat bully* utilizes priority and message absence as election means. When a backup process detects that the primary process does not send heartbeats, the detecting backup processes send a heartbeat requesting all higher priority processes to send a heartbeat, i.e., the backup process sends a reveal request heartbeat. If the process sending the reveal request heartbeat does not observe a heartbeat from a higher priority process within a specified period, the reveal heartbeat sending process is the new primary process.

We describe the details of *Heartbeat bully* with two state machines - one for the failure detection and one for the role selection, starting with the syntax used.

The transitions edge labels follow the format: $Trigger$/$Operation$/$Event$. An edge can have multiple labels representing alternative transitions separated by a linebreak.

- $Trigger$. Trigger of the transition, typically an internal trigger, such as the expiration of a timer, or external event or a command. External means that the event or command is external to the state machine described, for example, an event generated by the failure detection, but triggered on by the role selection, is external to the role selection.
- $Operation$. Operation to perform synchronous to the state transition.
- $Event$. Event raised synchronous to the state transition.

The following prefix and infix are used:

TABLE I
Heartbeat message fields.

Name	Description
BecomeSupervised (BS)	When set (value different than zero), and the value is equal to the identity of the receiving processes, this process should become the supervised process.
Priority	The priority of the heartbeat sender.
TieBreaker	A unique ID. Needed in case two priorities are equal, to break the tie and give precedence. Can for example be a system wide unique process ID.
Reveal	When set (value different than zero) all processes with higher priority than the sender, should send a heartbeat.

- Cmd. Prefix that denotes a command. A command is an external trigger to the state machine, issued by the state machine user. For example, the role selection state machine gives commands to the failure detection state machine.
- Ev. Prefix that denotes an event.
- Fd. Infix that denotes the failure detection state machine.
- It. Prefix that denotes an internal trigger, such as the expiration of a timer.
- Op. Prefix that denotes an operation.
- Rs. Infix that denotes the role selection state machine.

The priority handling is central; it is the means to decide which process should back away and provide precedence to another. The $TieBreaker$, see Table I, provides a priority tie-breaker to guarantee that all processes have different precedence. It is allowing dynamic properties, such as network connectivity, CPU-load, etc. to reflect the priority so that a more "fit" process gets precedence (higher priority). The $TieBreaker$ ensures that one process has higher priority, even if the $Priority$ values are equal.

A. Failure detection

The primary process cyclically sends a multicast heartbeat. We denote the period between two heartbeats $HbPeriod$. The supervising processes, i.e., the backup processes, checks every $HbPeriod$ if a heartbeat has been received. The maximum number missing heartbeats tolerated is $HbMissingMax$. $HbTmo$ is the time until the supervised process is assumed to have failed:

$$HbTmo = HbPeriod \cdot HbMissingMax \tag{2}$$

A supervising process resets the $HbPeriod$ timeout upon receiving a heartbeat. Two different $HbPeriod$, one for the sender (supervised process) and one for the receiving (supervising processes) is also an alternative, where $HbPeriodSnd$ denotes the $HbPeriod$ in the supervised process, the sender of the heartbeats. $HbPeriodRcv$ denotes the $HbPeriod$ in the receiving process, the supervising process. We need to ensure

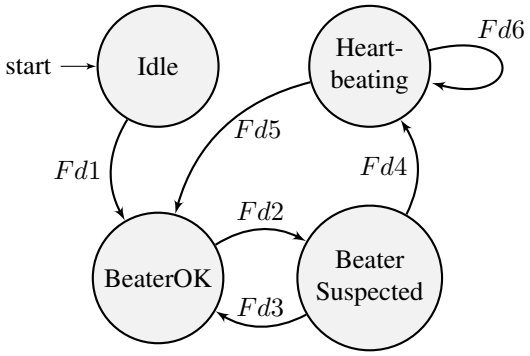

Fig. 1. Failure detection state machine.

that $HbPeriodSnd < HbPeriodRcv$ to avoid false positives failure indications. If $HbPeriodSnd = HbPeriodRcv$, i.e., the same $HbPeriod$, $HbMissingMax > 1$ ensure $HbPeriodRcv > HbPeriodSnd$. To avoid real numbers, we use $HbMissingMax = 2$.

Fig. 1 shows the failure detection state machine. The supervising processes remain in state $BeaterOK$ until a supervised process is suspected. An expiration of $HbTmo$ means that the supervised process is failure suspected. The failure detector raises the $EvFdSilence$ event. The role selection state machine, see Fig. 2, reacts to the $EvFdSilence$ and issues the command $CmdFdSupMeReveal$, that makes the failure detector enter the $Heartbeating$ state, and send a heartbeat with $Reveal$ set. $Reveal$ triggers all processes with higher priority to send a heartbeat.

1) States:

- $Idle$. Represents the initial/idle state, failure detection not active.
- $BeaterOK$. Supervising state where the supervision is active and receiving heartbeats from the supervised process.
- $BeaterSuspected$. The supervised process is suspected. There are three reasons for entering this state; $HbTmo$ has expired, received a heartbeat requesting this process to become supervised, or received a heartbeat with the $Reveal$ set from a lower priority process.
- $Heartbeating$. The state of the process being supervised. In this state, heartbeats are sent every $HbPeriod$.

2) Commands: The commands that can be given to the failure detection are the following:

- $CmdFdSupOther$ - start the supervision, failure detection, of the primary process.
- $CmdFdSupMe$ - become the supervised process.
- $CmdFdSupMeReveal$ - similar to $CmdFdSupMe$, sends a heartbeat with the $Reveal$ set instead of a regular heartbeat upon the transition to state $Heartbeating$.
- $CmdFdPassOn$ - sends a heartbeat with the BS field set. In other words, it passes on the heartbeating to another process, pointed out by the BS field.
- $CmdFdStop$ - stops the failure detection, transition back to $Idle$ state, left out of Fig. 1 for simplicity reasons.

3) Internal trigger: Trigger events internal to the failure detector, such as the expiration of a timeout timer or received heartbeat. Heartbeats are considered internal failure detector triggers since it is the sender and receiver of heartbeats.

- $ItHbTmo$ - $HbTmo$ expired, no heartbeat received within the $HbTmo$ timeout time.
- $ItLoPrRevealReqHb$ - received a heartbeat with the $Reveal$ set from a process with lower priority.
- $ItReqBeSupHb$ - received a heartbeat with the BS field set to the identity of this process.
- $ItHbLoPrio$ - received a heartbeat from a lower priority process when in $Heartbeating$ state.
- $ItHbHiPrio$ - a heartbeat received in $Heartbeating$ state from a process with higher priority.

4) Operations: Failure detector operations performed synchronously to a state transition.

- $OpSendHb$ - sends a heartbeat message.
- $OpSendHbReveal$ - sends a heartbeat message with the $Reveal$ set.
- $OpSendHbPassOn$ - sends a heartbeat message with the BS field set to the identity of the process that is to become the supervised process.

5) Events: Events raised by the failure detector.

- $EvFdSilence$ - the supervised process is silent, no heartbeat received within the $HbTmo$ period.
- $EvFdReveal$ - received heartbeat message with the $Reveal$ set, i.e., received a reveal request from a process with lower priority.
- $EvFdReqSup$ - the supervised process has requested this process to become the supervised process, i.e., to become the primary.
- $EvFdHiPrio$ - received a heartbeat, in state $Heartbeating$, from a higher priority processes.
- $EvFdLowPrio$ - received a heartbeat, in state $Heartbeating$, from a lower priority process.

6) Edge labels: The labels indicated in Fig. 1 are generic labels that are used for the sake of the presentation. In practice, they are expanded following the $Trigger/Operation/Event$ format as:

- $Fd1$: $CmdFdSupOther/-/-$
- $Fd2$: $ItHbTmo/-/EvFdSilence$
  $ItLoPrRevealReqHb/-/EVFdReveal$
  $ItReqBeSupHb/-/EvFdReqSup$
- $Fd3$: $CmdFdSupOther/-/$
- $Fd4$: $CmdFdSupMe/OpSendHb/-$
  $CmdFdSupMeReveal/OpSendHbReveal/-$
- $Fd5$: $ItHbHiPrio/-/EvFdHiPrio$
  $CmdFdPassOn/OpSendHbPassOn/-$
- $Fd6$: $EvFdLowPrio/-/-$

B. Role selection algorithm

The role selection utilizes the failure detection to achieve a deterministic role selection negotiation. The central part of the algorithm is the $Prospect$ state interaction with the failure detection, and the $Reveal$ handling. The $Reveal$ forces all

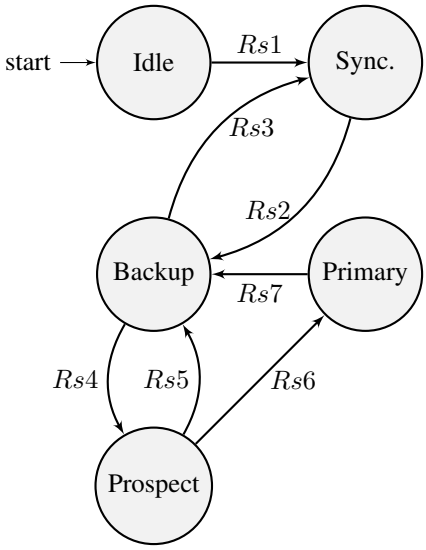

Fig. 2. Role selection state machine.

process with higher priority to reveal themselves and enter $Prospect$ state. Only the process with the highest priority, $TieBreaker$ included, will transition to the $Primary$ state. The lower priority processes will transition back to, or never leave, $Backup$ state.

1) States:

- $Idle$. Idle state, role selection is not active.
- $Sync$. The process is not synchronized with the primary process, i.e., not ready to take over. The state does not contribute to the algorithm per se - it exemplifies the broader context.
- $Backup$. The process is synchronized with the primary and ready to take over. In case no primary to synchronize with, the process is assumed synchronized.
- $Prospect$. Entered when this process is a primary prospect. When entering the $Prospect$ state due to a silent primary, the role selection instructs the failure detector to send a heartbeat with the $Reveal$ set immediately. Forcing all other processes with higher identity to reveal themselves. $Prospect$ state is left when a higher priority process reveals itself, i.e., receiving event $EvFdHiPrio$ from the failure detector, or upon the $PrTmo$ (prospect timeout) expiration. The $PrTmo$ time needs to be large enough to guarantee that all processes have reacted on the heartbeat with $Reveal$ set, i.e., allowing them to reveal themselves if they have higher priority. Using the same arguments as for $HbMissingMax$ in Section IV-A, we get.

$$PrTmo = 2 \cdot HbPeriod \tag{3}$$

  In the case of unreliable communication links, a longer $PrTmo$ is suitable. The $PrTmo$ period should, in that case, contain multiple resend heartbeats with the $Reveal$ set.
- $Primary$. Entered when ensured that there is no other process of higher priority, or when requested to become primary. In case a partitioning has occurred, each partition has a process in $Primary$ state. When the problems causing the partitioning vanishes, the two partitions will merge, the lower priority primary will receive a $EvFdHiPrio$ event and back away. The higher priority primary can receive a $EvFdLowPrio$ event and remain as primary.

2) Commands: The commands provided to the role selection user, for example, a process providing a control engine execution service.

- $CmdRsStart$ - start role selection.
- $CmdRsBackup$ - enter the Backup state, backup is synchronized with the primary (or no primary detected).
- $CmdRsNBackup$ - leave the Backup state, backup is out of synchronization with the primary (or some other reason for leaving the backup state).
- $CmdRsPassOn$ - pass on the primary role to another process.

3) Internal trigger: The role selection have one internal trigger, the expiration of the $PrTmo$ timeout.

- $ItRsProspectTmo$ - $PrTmo$ timeout expired, no higher priority process observed within the $PrTmo$ time.

4) Operations: Issuing failure detector commands are the only operations performed.

5) Events: The events raised by the role selection.

- $EvRsBackup$ - raised on the transition to the $Backup$ state.
- $EvRsNBackup$ - raised on the transition to the $Synchronizing$ state.
- $EvRsPrimary$ - raised on the transition to the $Primary$ state.
- $EvRsLowPrio$ - raised when another lower priority primary is detected. In a healthy setup, these events should not come unless there is partition merge, and in that case, it should be at most one.

6) Edge labels: The labels indicated in Fig. 2 are generic labels they are expanded following the $Trigger/Operation/Event$ format as:

- $Rs1$ - $CmdRsStart/-/-$
- $Rs2$ - $CmdRsBackup/CmdFdSupOther/EvRsBackup$
- $Rs3$ - $CmdRsNBackup/CmdFdStop/EvRsNBackup$
- $Rs4$ - $EvFdSilence/CmdFdSupMeReveal/-$ $EvFdRev/CmdFdSupMeReveal/-$ $EvFdReqSup/CmdFdSupMe/-$
- $Rs5$ - $EvFdHiPr/-/-$
- $Rs6$ - $ItRsProspectTmo/-/EvRsPrimary$
- $Rs7$ - $EvFdHiPr/-/EvRsBackup$ $CmRsPassOn/CmdFdPassOn/EvRsBackup$

C. Properties

In this section we present the properties of *Heartbeat bully*, for each of the requirements from Section III. In Section V we present the UPPAAL model and the verification queries used for validating that the properties hold.

1) Failover time: The failure detection and the role selection time is constant and independent of the number of processes. The time from silence until the selection of the highest priority backup as the new primary is the FoT (failover time), the maximum and minimum FoT are given by the equation:

$$FoT = \begin{cases} PrTmo + HbTmo - HbPeriod & \text{Min} \\ PrTmo + HbTmo & \text{Max} \end{cases} \quad (4)$$

The shortest failover time (Min) occurs when the supervised process fails just before $HbPeriod$ expires at the supervising process. The supervised process reset the $HbPeriod$ timeout upon receiving a heartbeat, hence the takeover time is longest when the supervised process fails just after sending a heartbeat.

2) Failover determinism: The priority handling and the reveal handling in the $Prospect$ state of the role selection provides a deterministic take over order.

3) Switchover: The BS field in the heartbeat message, see Table I, provide the switchover functionality. When a backup process P_i failure detector observes that BS contains the identity of P_i, it raises the $EvFdReqSup$ to the role selection, which enters the $Prospect$ state. Neither the Bully algorithm nor FBA have support for commanded leadership pass on.

4) Clinging: Heartbeat bully must pass through the $Backup$ role selection state, and the corresponding failure detector state $BeaterOK$, before it can transition to the $Primary$ state. In these states, it listens for an existing primary, which ensures that the primary process running, will remain the primary, even if a higher priority process recovers.

5) Single primary: The clinging and the failover determinism ensure that there will be at most one primary at any given time. A Byzantine process or communication channel failure is the only exception.

6) Number of messages: A typical property used to distinguish different leader election algorithms efficiency is the number of messages needed to elect a new leader upon leader failure. For example, the Bully algorithm [10] requires $\mathcal{O}(n^2)$, and the FBA [11] requires $\mathcal{O}(n)$ messages, where n is the number of potential leader processes. *Heartbeat bully* requires the maximum number of messages when the lowest priority process reacts first, i.e., it timeouts before the higher priority processes, and sends out a reveal heartbeat to all other processes. All processes, except the failed former primary process and the sending process, see the reveal heartbeat and also send out a reveal heartbeat to all other processes. Resulting in a total number of heartbeats of $hb = (n-1)^2$ to select the new primary, where n is the number of processes in the redundant configuration. The first glance suggests the same message performance as the Bully algorithm and worse than FBA. *Heartbeat bully* uses the heartbeat, and when the primary process fails, it stops sending heartbeats. The number of heartbeats sent if there would not have been a failure is equivalent to the number of backup processes multiplied with the failover time, FoT.

$$hbNrm = (n-1) \cdot FoT \quad (5)$$

If considering the failure detection period, the equation below describes the additional number of heartbeats resulting from the election:

$$\Delta hb = \begin{cases} (n-1)^2 - (hbNrm) & \text{if } hbNrm < (n-1)^2 \\ 0 & \text{if } hbNrm \geq (n-1)^2 \end{cases} \quad (6)$$

When using the failover time from Eq. 4, $3 \leq FoT \leq 4$, we get a $hbNrm$ range between:

$$hbNrm = \begin{cases} (n-1) \cdot 3 & \text{Min} \\ (n-1) \cdot 4 & \text{Max} \end{cases} \quad (7)$$

Using the smallest number of heartbeats that would have been sent in the normal case, $hbNrm_{min}$, the difference in heartbeats compared to a normal period are:

$$\Delta hb = \begin{cases} (n-1)^2 - hbNrm_{min} & \text{if } n > 4 \\ 0 & \text{if } n \leq 4 \end{cases} \quad (8)$$

When considering the FoT period, there are no additional messages needed to elect a new primary, compared to a normal run, for redundancy configurations containing up to four processes, and that is the worst case. The minimum number of heartbeats to elect a new primary occurs when the highest priority process reacts first and all the other processes see that heartbeat before timing out, i.e., $\Delta hb = n$. In the best case, when considering the FoT period the additional number of messages due to the election process is 0, regardless of the number of backup processes.

V. Formal method verification

A. Model

We use the model checking tool UPPAAL[1]. In UPPAAL, a *template* model each timed automata, and instantiated templates build the network of timed automata [5]. The following are the templates we use.

- ComChannelRx. We model communication between the processes with a simplistic communication channel. The channel consists of two templates. The ComChannelTx below, and this, the ComChannelRx, that models the receiver.
- ComChannelTx. Transmitting part of a communication channel.
- FailureDetector. UPPAAL model for the failure detector shown in Fig. 1.
- RoleSelection. UPPAAL model for the role selection shown in Fig. 2.
- UsingService. An example service that uses the role selection. The UsingService locations used in the verification queries are Idle, Primary, Backup, Passover and PrimaryWithTxComFailure. The location name serve as a description of the location, the details are in the model [27].

[1]The developed model is available for download https://github.com/Burne77a/HeartbeatBully

The model is configurable by changing the value of constants, such as changing the startup order of the processes and enabling communication failure triggering.

B. Verification

UPPAAL provides a verifier and query language to express and check the requirements [5]. The following subsection describes the verification queries used. We are using four, 1oo4, redundant processes. Zero identifies the process with the lowest identity and priority and three the process with the highest identity and priority.

1) Failover time: The process with the highest priority is the initial primary. We simulate a transmission error on the primary. The global clock $gTakeOverTime$ measures the takeover time. The maximum and minimum value of $gTakeOverTime$ when entering location $PrimaryU\_$ gives the shortest and longest takeover time. $PrimaryU\_$ is an urgent location, i.e., the transition from $PrimaryU\_$ to $Primary$ is instant, see UPPAAL turtorial and the model for details [5], [27]. When UsingService transitions to location $PrimaryWithTxComFailure$, its ComChannelTx is commanded to stop forwarding messages. The requirement checking function inf, short for infima, retrieves the shortest failover time.

```
inf{
(UsingService(3).PrimaryWithTxComFailure ||
UsingService(3).Backup) &&
UsingService(2).PrimaryU_}: gTakeoverTime
```

The longest failover time is retrieved by replacing the infima function with suprema in the query above. Inserting a value of $2 \cdot HbPeriod$ for $ProspectTmo$ and $HbTmo$ in Eq. 4 gives:

$$3 \leq failoverTime \leq 4 \tag{9}$$

The result from the UPPAAL verification confirms the calculation, $gTakeOverTime$ is between three and four.

2) Failover determinism: The process with the highest priority is the initial primary. The UsingService triggers a communication error. The query below checks that process with identity zero becomes primary, that process with identity one and two never becomes primary, and that process with identity three becomes primary. Combined with the verification for the clinging, the below proves the failover determinism.

```
E<> (UsingService(0).Primary)
A[] (!UsingService(1).Primary)
A[] (!UsingService(2).Primary)
E<> (UsingService(3).Primary)
```

3) Switchover: We verify the switchover functionality by enabling the UsingService to pass along the primary role. The UsingService passes on the primary role to the process with an identity one higher than itself. If no higher processes identity exist, the process identity to pass along to is zero. We check that all processes become primary, with the following query, one for each process identity. All are satisfied, meaning that we have been able to pass along the primary role to all processes. We also verify this using the simulation, see Section V-C1.

```
E<> (UsingService(0).Primary)
```

With the two queries below, we check the minimum and maximum time without a primary when passing on the primary role.

```
sup{UsingService(0).Passover &&
UsingService(1).PrimaryU_}: gTakeoverTime
inf{UsingService(0).Passover &&
UsingService(1).PrimaryU_}: gTakeoverTime
```

We use UPPAAL channels [5] for modeling messaging, i.e., message transfer is instant. The time without a primary is $PrTmo$, which is two. The above query confirms that.

4) Clinging: We verify the clinging by letting the lowest priority process, process zero, startup first, and thereby become primary. None of the other processes should become primary. The query below checks that there is a path in which the UsingService with the lowest identity becomes primary, and the three other queries verify that none of the other processes becomes primary. Hence, the process with the lowest priority is clinging on to the primary role.

```
E<> UsingService(0).Primary
A[] (!UsingService(1).Primary)
A[] (!UsingService(2).Primary)
A[] (!UsingService(3).Primary)
```

5) Single primary: The below query verifies that there never are two primaries.

```
A[] forall(i : id_t) forall(j : id_t)
UsingService(i).Primary &&
UsingService(j).Primary
imply i==j
```

The above query, combined with the query below, proves that there is one primary and only one primary, i.e., we check that there is a path where there is a primary and the query above has verified that there never are two, or more, different primaries.

```
E<> UsingService(0).Primary
```

C. Simulation

UPPAAL simulation with plotting is a swift way to test an algorithm and get model behavior feedback [6].

1) Switchover: The process with the lowest identity and priority starts as primary and then passes the primary role, as described in Section V-B3. The query below run the simulation for 100 time units. Fig. 3 shows the result.

```
simulate [<=100]
{UsingService(0).Primary,
UsingService(1).Primary,
UsingService(2).Primary,
UsingService(3).Primary }
```

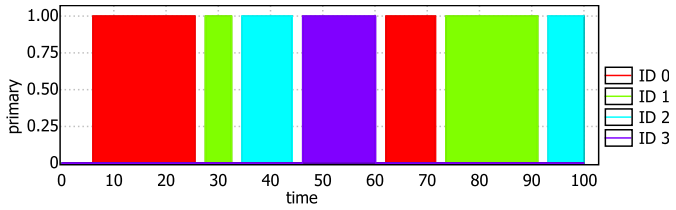

Fig. 3. Simulation of switchover in UPPAAL, showing the primary at each point in time.

2) Failover: The process with the highest priority start as primary, then the UsingService triggers communication failure by commanding the ComChannelTx to *Error* state. At that point, there is no working primary, until the process with identity two resumes. When the UsingService commands ComChannelTx to enter *OK* state again, and by that repairing the communication failure, there will be two primaries for a short while. Until the primary with the lowest priority back away, see Fig. 4.

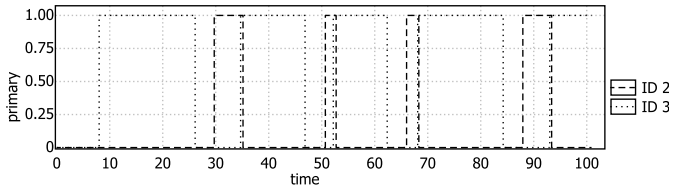

Fig. 4. Simulation of failover in UPPAAL, showing the process ID 2 taking over as primary when process ID 3 encounters communication failure.

VI. Conclusion and Future work

There exist many leader election algorithms for distributed systems, as well as many failure detection algorithms. Inspired by those algorithms, and with our problem formulation as input, we have deduced an algorithm capable of fulfilling the specified requirements. The algorithm provides relevant functionality for failure detection and redundancy role selection, for example, deterministic failover behavior with, known, upper bound takeover time. We verified the functions using a UPPAAL model. The model contains a service using the presented algorithm as well as a simplistic communication channel. With the UPPAAL model checking, we prove that the algorithm holds what we claim.

Several additions can be made to the model we developed, for example, probabilistic network delays and disturbances. Another natural continuation is to incorporate state transfer and look at the whole redundancy solution. We have tested the solution on a switched Ethernet network using UDP multicast heartbeat. Performance aspects of different networks, protocols, and security solutions are interesting continuations.

Industry 4.0 comes with more extensive utilization of virtualization and containerized solutions for cloud, fog, and edge computing. An evaluation of the suitability of this algorithm, or similar, in such a context, is a potential next step.

References

[1] R. Drath and A. Horch, "Industrie 4.0: Hit or hype? [industry forum]," *IEEE Industrial Electronics Magazine*, vol. 8, pp. 56–58, June 2014.
[2] P. Evans and M. Annunziata, "Industrial internet: Pushing the boundaries of minds and machines," *General Electric*, 01 2012.
[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct 2016.
[4] W. Steiner and S. Poledna, "Fog computing as enabler for the industrial internet of things," *e & i Elektrotechnik und Informationstechnik*, vol. 133, pp. 310–314, Nov 2016.
[5] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal 4.0," 2006.
[6] A. David, K. G. Larsen, A. Legay, M. Mikuundefinedionis, and D. B. Poulsen, "Uppaal smc tutorial," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, p. 397–415, Aug. 2015.
[7] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.
[8] M. van Steen and A. Tanenbaum, *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.
[9] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Commun. ACM*, vol. 22, pp. 281–283, May 1979.
[10] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Comput.*, vol. 31, pp. 48–59, Jan. 1982.
[11] S.-H. Lee and H. Choi, "The fast bully algorithm: For electing a coordinator process in distributed systems," in *Information Networking: Wireless Communications Technologies and Network Applications* (I. Chong, ed.), (Berlin, Heidelberg), pp. 609–622, Springer Berlin Heidelberg, 2002.
[12] M. Murshed and A. Allen, "Enhanced bully algorithm for leader node election in synchronous distributed systems," *Computers*, vol. 1, 06 2012.
[13] M. EffatParvar, N. Yazdani, M. EffatParvar, A. Dadlani, and A. Khonsari, "Improved algorithms for leader election in distributed systems," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 2, pp. V2–6–V2–10, April 2010.
[14] A. Arghavani, E. Ahmadi, and A. T. Haghighat, "Improved bully election algorithm in distributed systems," in *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia*, pp. 1–6, Nov 2011.
[15] M. Khan, N. Agarwal, S. Jaiswal, and J. A. Khan, "An announcer based bully election leader algorithm in distributed environment," in *Smart and Innovative Trends in Next Generation Computing Technologies* (P. Bhattacharyya, H. G. Sastry, V. Marriboyina, and R. Sharma, eds.), (Singapore), pp. 664–674, Springer Singapore, 2018.
[16] A. Biswas and A. Dutta, "A timer based leader election algorithm," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pp. 432–439, July 2016.
[17] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, Mar. 1996.
[18] N. Lynch, "A hundred impossibility proofs for distributed computing," in *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, (New York, NY, USA), pp. 1–28, ACM, 1989.
[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.
[20] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "A new adaptive accrual failure detector for dependable distributed systems," in *In ACM Symposium on Applied Computing (SAC 2007*, pp. 551–555, 2007.
[21] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "A lazy monitoring approach for heartbeat-style failure detectors," *2008 Third International Conference on Availability, Reliability and Security*, pp. 404–409, 2008.
[22] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," in *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing*, pp. 146–153, Dec 2001.
[23] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "A new adaptive accrual failure detector for dependable distributed systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, (New York, NY, USA), pp. 551–555, ACM, 2007.
[24] M. G. Gouda and T. M. McGuire, "Accelerated heartbeat protocols," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, pp. 202–209, May 1998.
[25] D. Dzung, R. Guerraoui, D. Kozhaya, and Y. Pignolet, "Never say never – probabilistic and temporal failure detectors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 679–688, May 2016.
[26] Wei Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, vol. 51, pp. 13–32, Jan 2002.
[27] "UPPAAL model of Heartbeat bully." https://github.com/Burne77a/HeartbeatBully. Accessed: 2020-07-19.