Reliable Control through Wireless Networks

Maxime Louvel, François Pacull, Maria Isabel Vergara-Gallego

Reliable Control through Wireless Networks

Maxime Louvel, François Pacull, Maria Isabel Vergara-Gallego
Univ. of Grenoble Alpes
CEA, LETI, MINATEC
Email: FirstName.LastName@cea.fr

***Abstract*—Reliability is one of the biggest challenges when using Wireless Sensor-Actuator Networks in control systems. This paper exploits the transactional guarantees offered by the LINC coordination environment to provide reliability and robustness in wireless control systems.**

First, LINC transactions were embedded in the mirco-controllers to deal with possible communication errors, faulty devices, and concurrent access to the devices. Then, an active replication mechanism was provided so that the system can be correctly recovered from hardware and communication failures.

A case study of a ball and plate problem is detailed. The plate is lift up and down by three motors. Each motor is controlled by a micro-controller communicating in wireless with the system controller.

I. Introduction

Wireless Sensor-Actuator Networks (WSANs) are being seriously considered to improve classical control applications such as process and building automation in the so called industrial IoT or Industry 4.0. Indeed, compared to classical wired technologies, wireless technologies are cheaper and easier to deploy and maintain. However, the wireless communication channel has an unreliable nature. Data will be lost or corrupted at some point, for instance, due to interference, obstacles, or other wireless communications.

Despite the efforts to improve communication error resilience [1] in WSANs, end-to-end data delivery is not guaranteed. In addition, variation of channel condition in industrial environments may further reduce the reliability of communication. In such applications, if data is lost (i.e. a command to an actuator), the system could go to an unsafe state. Let's consider the example of a room ventilated with a Heating Ventilating and Air Conditioning (HVAC) system. To save energy a window may be opened under some conditions. If the window fails to open and the HVAC is stopped the room will not be ventilated anymore.

Failures and errors in WSANs cannot be removed. Adding WSAN in industrial control systems requires to handle such errors properly. However, this error management must not add extra complexity in the system. Two types of failures can be distinguished: failure of the controlled system and failure of the controller.

This paper describes how the coordination environment LINC [2] is used to answer these challenges. LINC provides a high level rule based language to manage the complexity of WSANs. In addition LINC relies on a transactional engine to ensure the system consistency. In this work LINC has been extended to provide transactional guarantees up to the sensor/actuator level to handle failures in the controlled systems or the communication. This paper also details how LINC has been extended to provide active replication of the controllers.

This paper is organized as follows. After introducing LINC in Section II, Section III describes the proposed solution for reliability. Then, Section IV describes a demonstrator of a ball-and-plate control problem implemented with wireless communicating devices (ZigBee and Wi-Fi). Finally, Section V summarizes some related works and Section VI concludes the paper.

II. LINC Overview

This section briefly describes the coordination environment LINC to make the paper self-contained. Further details can be found in [2]. Inspired by Linda [3], the abstraction layer of LINC is an associative memory implemented as a distributed set of bags. Each bag contains resources shaped as tuples. Bags are accessed through three operations:

- `rd()`: takes a partially instantiated tuple as input parameter and returns from the bag a stream of fully instantiated tuples matching the given input pattern;
- `put()`: takes a fully instantiated tuple as input parameter and inserts it in the bag;
- `get()`: takes a fully instantiated tuple as input parameter, verifies if a matching resource exists in the bag and consumes it in an atomic way.

A. Coordination rules

The three operations `rd()`, `get()` and `put()` are used within production rules [4]. A production rule is composed of a precondition phase and a performance phase.

Precondition phase: The precondition phase is a sequence of `rd()` operations which detect or wait for the presence of resources bags. In the precondition phase:

- the output fields of a `rd()` operation can be used to define input fields of subsequent `rd()` operations;
- a `rd()` is blocked until at least one resource corresponding to the input pattern is available.

Performance phase: The performance phase of a LINC rule combines the three `rd()`, `get()` and `put()` operations to respectively verify that some resources are present (e.g. sensor values), consume some resources (e.g. events that have been handled), and insert new resources (e.g. to command actuators).

In this phase, the operations are embedded in one or multiple *distributed transactions* [5] executed sequentially. Every transaction executes a set operations in an *atomic manner*, i.e. all the operations are successfully done or none is executed. Thus LINC guarantees that:

- conditions responsible for firing the rule (precondition) are still valid at the time of the performance phase completion (e.g. a sensor value that has changed or an event that has been consumed by another rule);
- all bags involved in the transaction are accessible (i.e. no communication error occurred);
- all commands have been processed successfully.

Transactions are implemented using a two-phase commit. The first phase checks that all the actions involved in the transactions can be executed (e.g. resources are available, bags are accessible, and actuator commands are valid). In this phase, all the resources are locked for the transaction. Any other transaction trying to access a resource that has been reserved will be canceled and retried later. If no action failed in the first phase, in the second phase the actions are actually performed (i.e. resources are consumed and actuator commands are executed). If one action failed in the first phase the second phase is triggered to release the resources locked in the first phase.

LINC rules are executed by dedicated components called coordinators. A rule can be executed by several coordinators which may run in different machines or different networks to provide redundancy. Thanks to the transactions, the same rule may safely be executed by several coordinators. If, for instance, the same resource or the same actuator are accessed by multiple coordinators, only the fist coordinator that locks the resources will perform the transaction. The other coordinators will fail when accessing the resources and the transaction will be canceled and retried later. If the first coordinator successfully executes the transaction, the other ones will fail because resources have been removed from bags. If the first coordinator fails (e.g. power or network failure in the machine hosting the coordinator), the other ones will try to execute rule.

III. Reliable Wireless Control using LINC

This section describes an approach that, based on LINC, provides reliability on top of a WSAN. First, it describes how the LINC transactional protocol was integrated with the wireless devices; then, it describes how to introduce redundancy to the system by implementing an active replication mechanism.

A. Embedding Transactions in Sensors/Actuators

LINC relies on distributed transactions to enforce consistency of the system. Transactions guarantee that all (or none) actions of a group of actions are correctly executed. Therefore, at the end of a transaction it is possible to guarantee that no communication or hardware errors happened. In a scenario integrating wireless sensors and actuators, to ensure the correct execution of a transaction, these wireless devices must be able to behave in a transactional way. With this purpose, the LINC transactional protocol was embedded in the wireless devices; therefore, these devices can be accessed using LINC rules as if they were a standard LINC bag. The introduction of protocol-aware sensors/actuators was already proposed and partially implemented in a previous work [6]. This papers goes a step further, and implements full two-phase commit transactions.

To make WSAN nodes transactional, the main objective is to enable the coordinators to access the WSAN nodes through LINC rules. For that, a set of primitives at both the *coordinator* side and the *WSAN nodes* side must be implemented. Following, the proposed solution and its implementation are described; the most important extensions introduced by this work are highlighted.

The LINC protocol is provided by two components: the *stub* on the coordinator side and the *skeleton* on the WSAN nodes side. Once both sides have been implemented for a WSAN, the sensors and actuators can be manipulated through LINC rules as any other element in the application.

1) The Stub: To communicate with the WSAN nodes, coordinators must know how to access the WSAN nodes. For that, a piece of code called the *Stub* is available. The *Stub* provides all the primitives, invoked by LINC coordinators, to access resources in a bag. In a LINC application, a special object, called the *NameServer*, keeps information to construct a *Stub* on any bag belonging to the application.

When a *coordinator* wants to access a WSAN node, it asks to the *NameServer* information regarding the *Stub* associated to the WSAN node. Then, the *coordinator* can call the *Stub* primitives to communicate with the nodes with its network protocol. The *Stub* executes the proper methods to encode/decode frames to/from the WSAN and receive/transmit them from/to the desired nodes in the network.

Transmitted frames contain information such as the operation to be executed, the coordinator identifier, the transaction identifier, the bag to be accessed, the resource to be manipulated, and so on. Received frames contain the result of the operation. For each operation, the *Stub* encodes and sends the frame, and decodes the response from the node, to be used by the *coordinator*.

2) The Skeleton: The skeleton are all the primitives, implemented at the WSAN node side, to get the protocol working. The nodes primitives are called either in the precondition phase or in the performance phase.

Precondition Phase: The precondition phase detects the presence of resources. Implementing the precondition of a rule, requires implementing three basic operations:

- `Openstream`: this operation opens a stream, associated with a coordinator, that matches a given pattern. For instance, an openstream with the pattern `("gpio_1", "*")`, will open a stream associated to the current state of the `gpio_1` of the microcontroller.
- `Read`: after opening a stream, the coordinator can invoke the `read` operation, to actually retrieve the desired value (i.e. the state of the `gpio_1`). This `read` operation on a bag can be either blocking or non-blocking. In blocking mode, the coordinator is blocked when there is no matching resource in the bag (i.e. either no resource

matches or they all have been returned for this stream). When a new resource is added in the bag, if it matches the stream, the coordinator is unblocked. In the non blocking mode, the read returns a `no more resource` response to inform the coordinator that it is pointless to wait for new resources. The blocking mode is normally implemented by the skeleton. In this implementation, to save computation time in the nodes, the blocking `read` is implemented as a busy waiting loop by the stub. When the same resource is returned by the WSAN node, the stub waits for a configurable time and asks again to the WSAN node. Thus, from the coordinator point of view, the read is blocking. The notion of blocking/non-blocking `read` operations is an extension of [6], where only the non-blocking `read` operations were possible.

- `Closestream`: this operation closes an existing stream meaning no more read will be done for this stream.

Performance Phase: The performance phase corresponds to the execution of the two-phase commit transaction. Thus, it is formed of two phases: the *pre* and the *commit/abort* phases.

For the first phase, the following operations are provided:

- `pre_rd` and `pre_get`: these operations execute equivalent tasks. During these `pre*` operations, the resource, if existing, is locked. Then, other rule instances will not have access to this locked resource. If the resource is locked, this operation returns *busy*, telling the coordinator to retry the operation later. If the resource does not exist, the operation fails and the transaction is aborted by the coordinator.
- `pre_put`: when a `pre_put` operation is executed, the peripheral being accessed (i.e. the `gpio_1`) is locked. Thus, the value of the peripheral cannot be accessed or modified by any other rule instance. If the peripheral was locked by a previous `pre*` operation, the `pre_put` operation returns *busy*, telling the coordinator to retry the operation later. If the peripheral does not exist the operation fails and the transaction is aborted. Some resources may be read (`rd`) and modified (`put`) in the same transaction. For instance when reading and modifying the state of a gpio in a transaction. In this case, the `pre_put` operation returns *busy* because the resource is locked. To prevent this, the lock is ignored if both actions are executed in the same transaction (identified with the transaction identifier). This behavior is different from [6], where a `pre_put` operation does not lock the peripheral and no notion of transaction identifier is kept.

For the second phase of the transaction, the following operations are provided:

- `commit`: this operation is executed after a `pre_put` or a `pre_get` operation. In the first case, the operation (i.e. sending a value to a peripheral) is actually performed. In the second case, the desired resource is actually consumed (i.e. a local value in the node is consumed).
- `release`: this operation is executed when one action in the transaction fails. The `release` operation unlocks previously locked resources or peripherals (to rollback the transaction). The release operation is also used after a `pre_rd` operation. Indeed the `pre_rd` does not remove the resource from the bag, it is used only to validate the presence of the resource.

3) WSAN in LINC rules: Resources in the nodes correspond to values of any peripheral of the node microcontroller (e.g. GPIOs, ADCs, Timers, or an I2C/SPI device), a memory or local variable value, or any external device connected to the microcontroller. An application controlling devices in WSAN then consists of several LINC rules. The transactional protocol and reliability guarantees are completely transparent to the application. The application developer does not care about communication or hardware errors, as they are handled by the transaction.

Listing 1, shows an example of rule which accesses two different WSAN devices (*Arduino1* and *OpenPicus1*). In the example, line 1 reads the value of the `variable1` declared in the *OpenPicus1* device. Then, in the performance phase, line 4 determines if the value of `variable1` is still valid and, if this is the case, line 5 puts the value in the `spi_0` peripheral of the *Arduino1*. In the example, the action of sending a value through the `spi_0` is executed only if all the operations in the transaction succeed. All the operations that verify that the action can be performed are hidden to the application; the application developer only has to write three lines of code.

```
{*,!}["OpenPicus1", "Sensors"].rd("variable1", value) &          1
...
{                                                                3
["OpenPicus1", "Sensors"].rd("variable1", value);
["Arduino1", "Actuators"].put("spi_0", value);                   5
}.
```

Listing 1: Example of rule accessing WSAN devices

B. Active replication

Active replication consists in replicating the control of the system. In LINC, the control is done with LINC rules. Hence, rules replication offers redundancy and fault tolerance in case one coordinator fails. Thanks to the transactions, the possibility of replicating a rule comes for free: it simply needs to be executed by several coordinators. LINC ensures that if several instances of a rule want to consume or access the same resource, only one rule succeeds. Indeed, the other instances will fail, since resources are consumed by the first rule to succeed.

LINC transactions are executed in a two-phase commit approach. If a coordinator fails outside of the execution of a transaction, nothing need to be done. Another coordinator will do the job. However, if a coordinator fails during the execution of a transaction, resources may remain locked, preventing other coordinators to execute properly. Two situations might block the system:

- *The coordinator is in the first phase of the transaction*: the coordinator has locked some resources in the nodes. These resources cannot be accessed by another coordinator. Thus, the locked resources must be released for the system to continue working.

- *The coordinator is in the second phase of the transaction*: the coordinator has locked all the resources, and it has confirmed (or released) some of the actions. In this case, the transaction must be finished.

The coordinator may be stopped or disconnected at any moment, so, it is not possible to guarantee that these two situations will not happen. To recover from these situations, coordinators can inform other coordinators about the rules they are executing. It is possible to inform only about a subset of rules (i.e. the most critical ones). Coordinators that communicate between them are called here *Tandem Coordinators*. Each coordinator informs its tandems when:

- it starts a transaction (i.e. the coordinator is in the first phase);
- it finishes the first phase with the status of the first phase (i.e. success or failed) (i.e. the coordinator is in the second phase);
- it finishes the second phase of a transaction (i.e. the coordinator has finished the transaction).

This information is exchanged by adding resources in a special bag of the tandem coordinator. When a resource is added in this bag, it triggers a rule in the tandem coordinator. This rule waits for a configurable time. This time is different regarding the type of application or system. After this configurable time, a transaction is triggered to read the same resource and finish the transaction of the first coordinator. Hence, if the first coordinator continued its transaction, the rule executed by the tandem fails (e.g. the resource `phase_1_started` has been replace by `done`). If the resource is still there, it means the first coordinator has failed. The transaction is thus finished by the tandem.

If the first coordinator fails in the first phase, the tandem has to release all the resources reserved by the transaction. To do this, the tandem asks all the bags involved to cancel locks associated to the transaction identifier of the failed transaction. If the first coordinator fails in the second phase, the tandem has to confirm (or release) all the operations according to the status of the first phase. The tandem can not know which operations have already been confirmed (or released). Thus it confirms (or releases) all of the operations.

A potential drawback of rules replication is that, as multiple coordinators try to communicate with the WSAN nodes simultaneously, contention, delays, and the probability of loosing information are increased. Thus, the communication reliability decreases as the number of coordinators increases. To overcome this channel contention problem, only one coordinator is kept active at a time. The other coordinators execute a (configurable) time-out mechanism to determine whether the active coordinator has failed. This time-out mechanism is implemented by a rule. First, the active coordinator periodically adds its time-stamp value in the `time_out` bag of its tandems. Adding this resource triggers a rule in the tandems, which waits for a configurable time. After this time passes, the rule checks if the same time-stamp resource still exists. If it has changed, this means that the active coordinator is still alive, otherwise it means that the active coordinator has stopped working. In the case of failure of an active coordinator, a second coordinator is activated. The new active coordinator may finish transactions started by the faulty coordinator and it will continue to control the system.

Making a coordinator the active one, is done by the presence of a resource in a dedicated bag of the coordinator object. In the precondition phase of the rule, the presence of the resource is tested. If the resource is not there, the rule does not execute until the resource is added. Listing 2 shows how the resource is read, from the bag `State` of the actual rule is executed.

```
{*,!}["CoordinatorA", "State"].rd("active") &
...
Coordination rule
...
```

Listing 2: Test if a coordinator is active

IV. Case Study: the Ball and Plate Example

This section presents a demonstrator of a ball and plate control problem implemented with LINC. Three motors are used to lift up and down a plate with a ball on top of it. The motors are moved step by step. To lift the plate, a command is sent to each motor, via a wireless link, to make it turn one step in the same direction. Due to delays in communication, it is not possible to guarantee that the three step movements are performed at the same time. Therefore, there is a small hole in the middle of the plate, to keep the ball in place when the motors do not move concurrently. This demonstrator can handle one step difference between the motors. However, with more than one step, the ball will fall from the plate.

This demonstration shows the ability of LINC to reliably control a system with unreliable devices and communication links. Despite no real-time guarantees are provided here, the consistency of the system is ensured.

Figure 1 shows the components and the architecture of the demonstrator. There are two *Tandem* coordinators, running on different devices for redundancy: a *Beaglebone Black*, and a *RaspberryPi B*. The coordinators execute the same rule that moves the three motors step-by-step, inside a transaction. Each motor is controlled by a wireless communicating device. The first motor is controlled by an *ArduinoUno*, which communicates through *ZigBee*. The other two motors are controlled by two *Openpicus* communicating through *Wi-Fi*. Both, *ArduinoUno* and *Openpicus* communicate using a predefined protocol stack (*ZigBee* and *Http over Wi-Fi* respectively). The LINC transactional protocol was implemented on top of these protocol stacks.

The coordinators can access the WSAN nodes using the special objects `object_arduino` and `object_openpicus_n`. The latter objects implement special *Stub*s to access nodes in the specific WSAN.

In addition, the demonstrator provides a lightweight interface used to monitor the state of the system. This interface is accessible from a web browser (e.g. from a tablet or a mobile phone). The interface application is also synchronized with the system. It is seen as another equipment and, if absent,

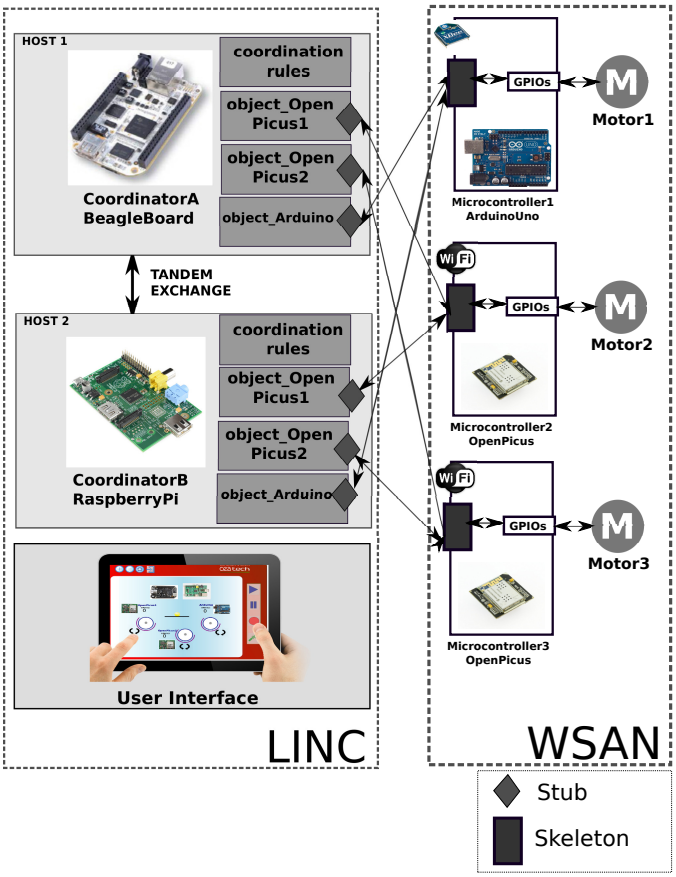

Fig. 1: Components of the demonstration

the system stops working. This demonstrates that software and hardware can be safely coordinated in a wireless and distributed system.

The case study supports the following errors:

- *One coordinator failure (hardware, software or communication)*: If the faulty coordinator is active, the second coordinator will detect the error, and it will continue executing the rule that moves the motors.
- *One or several nodes communication failure*: If a node becomes unreachable when used in a transaction, an error is raised to the coordinator by the stub. The coordinator will stop trying to execute this transaction for a configurable time and retry until it succeeds. The resources locked stay locked preventing any transactions to move the motors. When the node becomes available again (i.e. communication is working again), the transaction is finished by the coordinator.
- *One node hardware or software failure*: If a hardware or software fault occurs in a node, the transactions will stop as for a communication failure. However, in this case, the node needs to be restarted. As soon as the faulty node is reconnected, its state is forced to the current motor state, which can be retrieved from the other two nodes.
- *User interface failure (hardware, software or communication)*: The user interface is provided by a LINC object.

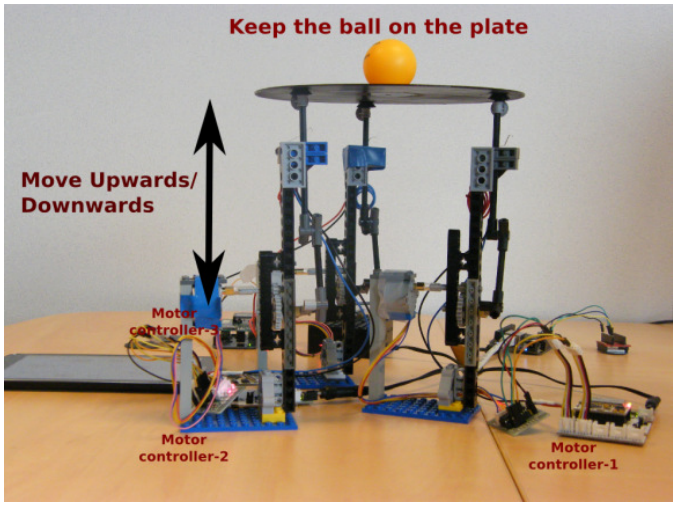

Fig. 2: Picture of the demonstration

Thus failures are handled exactly as the node failures.

In addition the following features are offered by LINC:

- *Coordinator restart*: A coordinator may be stopped at any time. It will stop properly (i.e. finish all the started transactions and stop). The other coordinator will become active. The first coordinator can then be started again on the same machine or on a different one.
- *Node restart*: Nodes can be safely restarted while the application is running (i.e. for a software update or node replacement). The restart will be handled as a hardware failure.

Figure 2 shows a picture of the demonstrator. It illustrates how the plate, with the ball on top of it, is moved upwards and downwards by the three motors.

```
{*,!}["OpenPicus2", "Actuators"].rd(step)&
INLINE_COMPUTE: new_step=str(int(step)+1) &                    2
::
{                                                              4
["Arduino1", "Actuators"].put("NStep", new_step);
["OpenPicus1", "Actuators"].put("NStep", new_step);            6
["OpenPicus2", "Actuators"].put("NStep", new_step);
["Interface", "steps"].put("steps", new_step);                 8
}.
```

Listing 3: Rule to control the movement of the three motors

Listing 3 shows the rule used to control the system. The precondition phase reads (line 1) the current step of one of the motors (i.e. the one connected to the *Openpicus2*). As the step values of all the devices are the same, reading the value of one of them is sufficient. According to the current step value, the new step value is computed (line 2). Then, the performance phase (after `::`) sends the commands to move the three motors (line 5 to 7). In the example, `NStep` is a customized peripheral that keeps a variable with the number of performed steps and that moves the motor when this variable is modified. Finally, the interface is updated (line 8).

During the first phase of the performance of Listing 3, all the `pre_put` operations are executed to lock the `NStep` resource in each node. If the communication with one node fails, or

the motor is not accessible, the transaction is rolled-back (all locked resources are released) and the position of the three motors does not change. If the first phase is successful, the second phase of the performance executes the `commit` operations and the actions are actually executed. If the first phase failed, the other operations are released and nothing happens. The put operations are sent in a sequential way. If the confirm operation of one of the motors does not work, the transaction will be blocked waiting for the device to be accessible. In this case it is possible that only one or two motors move, and the plate will not be completely horizontal. Therefore, the system must tolerate one step difference between the motors.

V. Related Work

The increasing interest in wireless technologies for industrial applications has motivated the development of algorithms and solutions to support this type of applications [7]. These solutions, implemented at the network and lower layers, aim at improving communication reliability, security, and delay. The work in this paper proposes an application-level solution for reliability, that can be integrated with lower level solutions, to further improve the reliability and enforce the consistence of the system. Similarly, Feng et al. [8] propose an application-level solution that relies on packet loss knowledge to keep controlling the system even when data is lost. Nevertheless, given the distributed nature of WSANs, guaranteeing a consistent state of the global system in this case is challenging.

When performing rules replication, the problem of resource locking when a coordinator fails must be addressed. This problem has been tackled in database access applications where transactions are common use. For instance, three-phase commit protocols [9] remove the blocking problem at the price of data exchange overhead and high complexity. For this reason, two-phase protocols are still preferred over three-phase protocols. Other solutions [10] propose the participants (in our case the WSAN nodes) to participate in the failure recovery procedure. Due to the highly constrained computational resources in WSANs, such approach is not a feasible solution for embedded devices. Manikandan et al. [11] propose the use of a *backup coordinator* and a *connection manager*. If the main coordinator fails, the connection manager detects the failure and transfers all transactions to the backup coordinator. However, if the connection manager fails, or it cannot access the faulty coordinator, the application will stop working. The use of *Tandem coordinators* do not require the intervention of a third-part entity; in this case, the coordinators decide in a distributed way if they should finish an ongoing transaction.

VI. Conclusions

This paper proposed a solution to build reliable control systems using Wireless Sensor-Actuator Networks (WSANs). The proposed approach provides reliability even under communication errors and hardware failures. The work relies on the coordination environment LINC. LINC provides distributed rule engines and transactional guarantees.

LINC primitives were embedded into the WSAN nodes, so that transactions can be performed when accessing sensors and actuators in the network. Besides, thanks to the use of transactions, LINC permits introducing replication of rules. An active replication mechanism was proposed to recover the system when there is a failure in one LINC component.

A case study of a ball and a plate is presented. The case study is composed of three step motors lifting up and down a plate with a ball. The motors are controlled by two types of embedded platforms: *Openpicus* and *ArduinoUno*. The first uses a WiFi communication and the latter Zigbee communication. The three motors are controlled by LINC components running in a *Raspberry PI* and a *Beaglebone Black* with active replication. The objective of keeping the ball on the plate is achieved even with communication and hardware failures. In addition a software user interface is also synchronized with the three motors. Thanks to the proposed approach, only 6 lines of application code have been written.

Acknowledgment

This work has funded by the Artemis ARROWHEAD (grant 332987) and the H2020 TOPas project (grant 676760).

References

[1] M. A. Mahmood, W. K. Seah, and I. Welch, "Reliability in wireless sensor networks: A survey and challenges ahead," *Computer Networks*, vol. 79, pp. 166 – 187, 2015.

[2] M. Louvel and F. Pacull, "Linc: A compact yet powerful coordination environment," in *Coordination Models and Languages*. Springer, 2014, pp. 83–98.

[3] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, pp. 444–458, 1989.

[4] T. Cooper and N. Wogrin, *Rule-based Programming with OPS5*. San Fransisco: Morgan Kaufmann, 1988, vol. 988.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. New York: Addison-wesley, 1987, vol. 370.

[6] H. Iris and F. Pacull, "Smart sensors and actuators: A question of discipline," *Sensors & Transducers Journal*, vol. 18, no. special Issue jan 2013, pp. 14–23, 2013.

[7] G. Zhao, "Wireless sensor networks for industrial process monitoring and control: A survey," *Network Protocols and Algorithms*, vol. 3, no. 1, pp. 46–63, 2011.

[8] F. Xia, Y.-C. Tian, Y. Li, and Y. Sun, "Wireless sensor/actuator network design for mobile control applications," *CoRR*, vol. abs/0806.1569, 2008.

[9] D. Skeen, "Nonblocking commit protocols." ACM Press, 1981, pp. 133–142.

[10] B. W. Lampson and D. B. Lomet, "A new presumed commit optimization for two phase commit," in *Proceedings of the 19th International Conference on Very Large Data Bases*, ser. VLDB '93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 630–640.

[11] R. S. V. Manikandan, R. Ravichandran and F. S. Francis, "An efficient non blocking two phase commit protocol for distributed transactions," *International Journal of Modern Engineering Research*, vol. 2, pp. 788–791, 2012.