Reliable Upgrade of Group Communication Software in Sensor Networks

Prasanna V. Krishnan[1]
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
pkrish@microsoft.com

Lui Sha
Univ. of Illinois at Urbana-Champaign
1304, W. Springfield Avenue
Urbana, IL 61801
lrs@cs.uiuc.edu

Kirill Mechitov
Univ. of Illinois at Urbana-Champaign
1304, W. Springfield Avenue
Urbana, IL 61801
mechitov@cs.uiuc.edu

***Abstract*-Communication is critical between nodes in wireless sensor networks. Upgrades to their communication software need to be done reliably because residual software errors in the new module can cause complete system failure. We present a software architecture, called cSimplex, which can reliably upgrade multicast-based group communication software in sensor networks. Errors in the new module are detected using statistical checks and a stability definition that we propose. Error recovery is done by switching to a well-tested, reliable safety module without any interruption in the functioning of the system. cSimplex has been implemented and demonstrated in a network of acoustic sensors with mobile robots functioning as base stations. Experimental results show that faults in the upgraded software can be detected with an accuracy of 99.71% on average. The architecture, which can be easily extended to other reliable upgrade problems, will facilitate a paradigm shift in system evolution from static design and extensive testing to reliable upgrades of critical communication components in networked systems, thus also enabling substantial savings in testing time and resources.**

I. INTRODUCTION

Communication is vital between nodes in a wireless sensor network because control is distributed and most common tasks require collaboration between nodes. For example, information exchange is needed to triangulate the location of a vehicle being tracked. Due to the diverse applications of sensor networks, as well as the need to optimize different constraints like energy, bandwidth *etc*., several multicast based routing protocols have been developed for sensor networks. Examples include proactive protocols like AMRoute, AMRIS, CAMP and MCEDAR, as well as, reactive protocols like AODV and ODMRP [3,4,5,6,7,8].

In addition, several multicast transport protocols have been developed not just for sensor networks, but for multicast applications in general, since no multicast transport protocol is generic or powerful enough for all types of multicast applications [2]. Scalable Reliable Multicast (SRM), Uniform Reliable Group Communication Protocol (URGC), Muse, and Log-Based Receiver-reliable Multicast (LBRM) [10,11] are some examples. As sensor networks and multicast applications evolve, communication protocols for sensor networks as well as multicast transport protocols are constantly changing and need to be upgraded frequently. This underscores the need for reliable upgrade of multicast communication software in general and multicast based group communication in wireless sensor networks in particular. A reliable upgrade mechanism is particularly important in sensor networks because:

(i) The cost of failed upgrades can be enormous in terms of system downtime and wasted resources.

(ii) Due to potential placement in remote or hostile locations, manual re-configuration in the face of an error may be expensive or even infeasible.

(iii) The upgrade is often to a new protocol that may contain software errors.

(iv) It is impossible to exhaustively test or formally verify all of the group communication software, and static testing in the laboratory does not guarantee reliability since the protocol may still contain residual errors that could bring down the network when deployed.

Statistics show that 50 to 90% of the costs associated with a software-intensive system are incurred from upgrades made to the original software [15] because software changes are error-prone. Hence there is a need for a rigorous and disciplined approach to ensuring reliability in upgrading the communication software in sensor networks. In this research, we propose an architecture called Simplex for Communication or cSimplex, that permits reliable upgrade of the multicast based group communication in sensor networks in spite of residual errors in the new software.

A.. Background and Related Work

There are two traditional approaches for software reliability, as outlined in [1]. One is the fault avoidance method using formal specification-verification methods and a rigorous software development process such as the DO 178B standard used by FAA for flight control software certification. Though these are powerful methods that allow for computer-controlled safety-critical systems such as

[1] This work was done when the author was a graduate student at UIUC.

This work was sponsored in part by the Office of Naval Research and in part by DARPA's Networked Embedded Software Technology (NEST) program.

flight control, they are expensive and can handle only modestly complex software. The trend towards using large networked system of systems based on commercial-off-the-shelf (COTS) components also makes the application of the fault avoidance method more difficult.

The second approach is software fault tolerance using diversity, *e.g*. N-version programming[17]. Here, different programmers build different versions of the same software with the idea that different designers and implementers will produce different errors. Therefore, when one system fails under a given set of circumstances, there is a high probability that the other will not. It is a widely held belief that diversity entails robustness. However, as shown in previous work on the Simplex Architecture [1], dividing resources for diversity could lead to either improved or reduced reliability, depending upon the architecture. The key to improving reliability is not the degree of diversity *per se*; rather it is the existence of a simple and reliable core component that can ensure the critical properties of the system, in spite of the faults of the complex software. We call this approach "using simplicity to control complexity". The Simplex Architecture was developed to support this approach.

Simplex was originally developed for reliable upgrade of control software in mission-critical real-time systems [1]. This work extends Simplex to the domain of communication software upgrade in sensor networks. Unlike control software which consists of deterministic loops, communication software poses new problems: a) it is more complex, since it is inherently a distributed system problem, and b) determining the stability of communication software is inherently statistical in nature.

Section 2 presents an overview of the cSimplex architecture. In Section 3, we define the concept of stability for multicast based group communication. Section 4 describes cSimplex's fault detection mechanism, which is based on the stability definition of the previous section. Finally, Section 5 discusses the experimental setup and results obtained.

II. Overview of cSimplex

cSimplex allows safe, online upgrade of group communication software that may contain residual errors, without compromising on reliability. As seen in Fig. 1, a reliable core 'cSimplex' component runs on each node in the network. The cSimplex components on peer nodes collaborate to manage online upgrade, error detection and recovery. In addition, the nodes have an experimental communication module as well as a simple, well-tested and hence reliable communication module for back up purposes. The latter is referred to as the safety module. Both communication modules have a system-level and an application-level component. The application-level

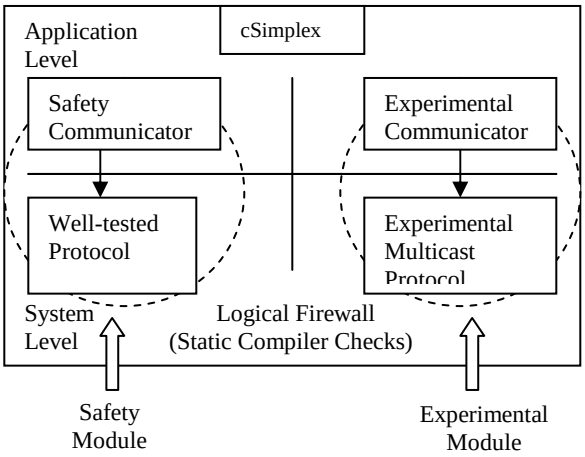

Fig. 1. Elements of the cSimplex Architecure

component, called communicator, implements application specific group communication using the infrastructure provided by the system-level protocol stack. For example, in a sensor network to monitor vehicle traffic, the application communicator periodically exchanges data with an application-determined set of neighbors using a system-level multicast protocol like AMRIS [3] or CAMP [5]. The advantage of this two-level architecture is that it allows for independently upgrading both the system and the application-level components.

The *experimental module* implements a new multicast group communication protocol, *e.g.* a higher performance protocol that is customized for the specific sensor network application. However, like all newly developed software, it often contains errors. The *safety module* is based on a *high assurance*, general-purpose protocol that is more robust but may be less efficient for the specific communication scenario. The goal of cSimplex is to detect software errors in the experimental module, based on the stability definition in this paper and user-provided application-specific requirements, and to maintain communication by switching to the safety module.

The application components of the communication modules subscribe to cSimplex and are written in such a way that they can be dynamically replaced without shutting down the system. This subscription-based online component replacement mechanism is a reuse of low-level Simplex architecture tools [23]. However, the system-level component can only be replaced offline, because doing this online would require the protocols to be implemented as user-downloadable kernel modules and the operating system to support kernel protection for dynamic kernel-level code insertion. Popular present day operating systems do not offer this capability. The SPIN microkernel [13], proposed by University of Washington, is an operating system that supports this using compiler and language assistance provided by Modula-3. However, developing a solution for kernel-level issues is outside the scope of this study.

The core cSimplex component executes at the application level, thus the architecture uses application-level checks to determine violations of the stability criteria. Errors in the system-level protocol stack are detected based on manifestations of these errors at the application-level. Some system-level errors, like those that cause routing loops, may not be evident directly at the application-level. These errors are detected indirectly, *e.g.* through delays introduced by the routing loops.

III. Communication System Stability

A system is said to be stable if: 1) errors are bounded within acceptable levels and 2) essential services are delivered. In the following, we will first define our fault model, then define the concept of group communication stability and use it to guide fault detection and error recovery.

A. Fault Model

The fault model for cSimplex targets software faults in both the system-level and the application-level components of the experimental module.

1) Software faults at the system level: System-level code has to meet the minimal requirement of memory safety. Ensuring that the system-level component of the experimental communicator is memory safe would require static compiler checks [22]. This is represented as a logical firewall in Fig. 1. cSimplex, on the other hand, targets semantic faults at the system level that violate basic multicast invariants. Semantic faults are caused by algorithmic errors and bugs in software design and implementation. An example of a semantic fault would be a software fault in the implementation of the group management module of the experimental multicast protocol, as a result of which a node receives a message that was not sent by a valid member of its multicast group.

2) Software faults at the application level: Since the application component of the experimental module runs as a separate process with its own address space, cSimplex gains process-level fault protection at no additional cost. Thus, our focus is on semantic faults in the application-level group communication that result in violations of application-specific invariants. For example, the safety communicator for a sensor network application meets application requirements with respect to message delays, but may not be energy-efficient. The experimental communicator uses a more energy-efficient communication scheme to share information with its peers, *e.g.* using metadata negotiation, as done in the SPIN protocol family [9], to avoid transmitting redundant data. The two communicators may use the same or different system-level protocol. Since the experimental communicator is newly developed, it may contain software faults, *e.g.* an error in the code that sends data periodically may cause messages to be delayed beyond the amount that the application can tolerate.

We assume that the upgrade software is written by a trusted entity, and hence do not deal with malicious attacks. Additional protection against system errors such as capability abuse and malicious code can be provided through a combination of operating system and compiler support [22].

cSimplex's fault model does not target faults other than software faults, *e.g.*, at the system level it does not target hardware or link failures. Similarly, at the application level it does not target faults like errors in application code besides that related to group communication. For example, in our experimental setup, we do not target software errors in the path-planning algorithm used by the robot or in the control software that moves the robot. Since the Simplex Architecture has been shown to work for control software, the aim of this work is restricted to the domain of communication software.

B. Stability for Multicast Communication

System correctness is defined as meeting all the specifications. System stability is defined as meeting a subset of requirements that are deemed to be critical. In other words, a system is said to be stable in the sense that critical services are delivered and errors are being kept at an acceptable level. The authors of [11] argue that unlike unicast where requirements for reliable, sequenced data delivery are fairly general, different multicast applications have widely different requirements for reliability. For example, some applications require totally ordered message delivery while others do not; some have one sender and many receivers, while for other applications all group members may be data sources. Thus it is not possible to define a single set of critical requirements for all multicast applications. Hence, we propose a two-level definition of stability for multicast consisting of basic and application-specific critical requirements.

The following are basic critical requirements of reliable multicast [12] that must be met irrespective of the application using the protocol:

- **Agreement**: All valid nodes receive the same set of messages. If a valid node receives a message m, then all valid nodes eventually receive m.
- **Validity**: All messages sent by valid nodes are received. If a valid node sends a message m, then all valid nodes eventually receive m.
- **Integrity**: No spurious messages are received. For any message m, every valid node receives m at most once, and only if m was previously sent by a valid sender.

In the context of group communication, we define the following as basic invariants that have to be satisfied for a protocol to be considered stable:

- Messages are received only from valid group members (Integrity).
- Spurious messages that were never sent by a valid sender are not accepted.
- Given that system requirements like bandwidth availability and infrastructure are met, the delay of a message does not exceed the application-specified tolerance due to software errors.
- Messages from a valid sender are not consistently delayed.
- Messages from a valid sender are not consistently lost.
- All valid recipients of messages sent out by valid senders should receive them without significant loss or delay.
- Messages are not corrupted due to software faults
- Messages are not duplicated by the communication software

In addition to these basic stability requirements, users can specify application-specific requirements:

1) Application-specific timeouts for data: Many sensor network applications have periodic sensor data flows. Thus they may specify a delay tolerance beyond which the data is no longer useful to the system.

2) Performance and error thresholds: The application specifies the level of errors that it is willing to tolerate. For example, in order to function properly, an application may require message losses to be below a maximum acceptable level.

3) Features specific to the application, *e.g.* ordering requirements like causal order, total order etc.

cSimplex allows an application developer to plug such application-specific checks into the decision module. For example, the acoustic tracking application in our experiments requires *most-recent reliable* multicast without any ordering requirements. Most-recent reliability is defined as reliable transmission where only the most recent data of a particular parameter is of interest [18]. The decision module for this application incorporates application requirements like tolerable delays and losses, but does not enforce ordering, since that is not an application requirement. Some other application that does require a specific ordering of messages, e.g. FIFO order, where messages are delivered in the same order that they were sent, can provide a suitable decision function that can be plugged into cSimplex.

The safety communication module for a given application meets both basic and application-specific stability requirements, although it may not be the most efficient protocol. Hence, cSimplex assumes that top-level applications can adapt to the level of service provided by the safety module and function acceptably if the experimental module fails. cSimplex will normally execute the experimental module, which is presumed to be more efficient; however, it will switch to the safety module if the experimental module violates any stability requirements. The default response to the violation of user specified requirements is to switch to the safety module. A user can override the default behavior by providing a specific error handling routine that is more appropriate for the given application.

IV. Fault Detection

Checking for stability of the experimental software is done in a completely distributed manner with no centralized control and coordination entity. This makes the architecture more scalable and reliable, as there is no single point of failure. This is important in sensor networks as node failures, e.g. due to battery depletion, are common.

As discussed in [11], single-point, purely sender-based control using ACK/NACK, as done in TCP, does not scale well for multicast. cSimplex on each node performs checks as both a sender and a receiver to detect faults in the experimental module. As the experimental results in the next section demonstrate, using a combination of sender-side and receiver-side stability checks both improves accuracy and decreases the latency of detecting errors in the communication modules.

As seen in Fig. 2, cSimplex consists of three main components: the Subscription Thread, the Data Thread and the Inter-cSimplex Message Thread.

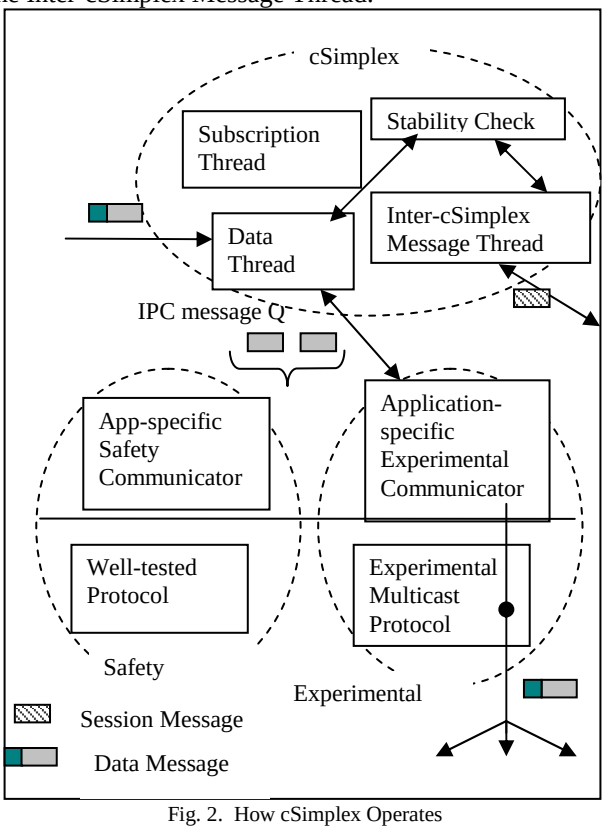

Fig. 2. How cSimplex Operates

cSimplex uses a publish-subscribe scheme so that application-level communicators can be dynamically replaced without shutting down the system. On startup, both safety and experimental communicators subscribe to cSimplex, a process handled by the Subscription Thread.

Data Messages exchanged between the communicators are read in by cSimplex and passed to the communicator after the appropriate checks have been performed. Semantics for creating the communication socket for incoming messages depends on the underlying multicast protocol used by the experimental communicator. In order to allow cSimplex to work with any protocol, the communicator creates the data socket and passes it to cSimplex as part of the subscription request.

As seen in Fig. 3, cSimplex adds header fields in the Data Messages to collect statistics for stability checks. After performing stability checks, cSimplex passes on the application data to the communicator. Based on incoming data from its peers, the communicator performs application-specific computations or takes action. It then sends out one or more Data Messages after cSimplex adds the header. Both the experimental and safety communicators follow this pattern of interaction. cSimplex uses timeout mechanisms at all steps in the interaction to prevent a deviant communicator from blocking the system indefinitely.

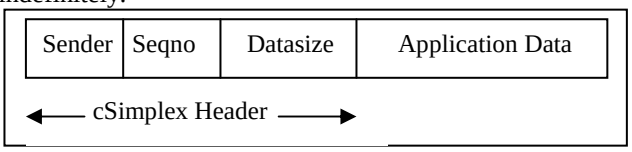

Fig. 3. Data Message

A. Mechanism of Checking for Stability

Based on literature surveys [19, 20, 14, 21], Table 1 enumerates common software faults that may occur in either the application-level communicator or in the system-level protocol implementation, along with their manifestations.

TABLE 1
COMMON FAULTS IN MULTICAST SOFTWARE

Software Fault	Manifestation
Malformed messages due to software errors	i. Messages with invalid sender ID; ii. Messages that were never sent by a valid sender.
Messages are dropped due to faults in the communicator's logic or in the system-level protocol implementation, e.g. in the routing software	Lost messages
Packet storms due to errors in protocol implementation. Multicast packets flood the network creating excessive traffic and degrading network performance.	i. Various patterns of message delays with different amounts of delay ii. Message losses
Errors in the group management module of the multicast protocol stack.	i. One or more valid group members do not receive *any* messages sent to the group ii. Messages delivered to a node that is not a valid group member
Improper data delivery due to error in the communicator or system protocol stack.	Random message losses and delays experienced by one or more members
Repeated message sends due to a bug in the application code that sends messages to peers.	Duplicate messages received at some or all nodes.
Routing loops due to an error in the multicast protocol stack. Dangerous since faulty nodes act as amplifiers of data packets, making several copies of the packets received. This can quickly exhaust bandwidth in all or a portion of the multicast tree.	i. Simultaneous duplication of messages at multiple nodes ii. Delayed messages: effect of routing loops as seen at the application level.
Local reception errors	Only some receivers do not receive a message

Keeping these common software faults in mind, we developed the following mechanism to detect errors using the stability definition. cSimplex on each node performs two broad categories of checks:

- Using fields in the cSimplex header of Data Messages exchanged between communicators. This is done by the Data Thread.
- Using periodic management messages, called Session Messages (Fig. 4), exchanged between peer cSimplex modules. This is done by the Inter-cSimplex Message Thread.

B. Checks based on Data Message Headers

The D*ata Thread on the receiver end* maintains the following statistics:

- Number of messages received from an invalid sender
- Number of messages from each sender that were lost, delayed beyond levels acceptable to the application, or duplicated.

The sender ID in Data Messages is used to determine if the message is from a valid group member. If a gap occurs in the sequence space of received messages, timers are started for the missing messages, to determine if they are lost or delayed.

C. Checks based on Session Messages

Since timers are started only when the Data Thread notices a gap in the sequence numbers of received Data Messages, it is possible that all the messages from a certain point till the end of a transmission are lost or delayed. As the receiver is not aware of the total number of messages being sent, these errors would go undetected. Also, if a receiver did not receive any messages from one or more senders, it would not be able to detect the error, as it does not see any gaps in the received sequence number space. Thus there is a need for stability checks on the sender side. To do this, the cSimplex components running on different nodes periodically exchange S*ession Messages* (Fig. 4), at

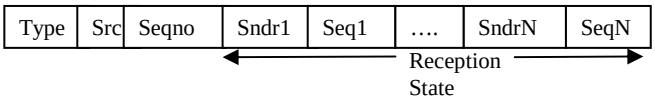

Fig. 4. Session Message

a low rate. These management messages are transparent to application-specific communicators.

In addition to Session Messages, peer cSimplex modules exchange other management messages at startup and prior to termination of a faulty experimental module. *Type* is used to distinguish between these management messages. *Source* identifies the node that sent out the Session Message. *Sequence numbers* of Session Messages are separate from sequence numbers of Data Messages. Session Message sequence numbers are not used for stability checks. They are only used by the recipient cSimplex to identify the most recent Session Message from a sender and use the information therein. R*eception state* indicates the highest sequence number D*ata Message* that the *source* of a Session Message has received from each of its senders. By looking at the reception states of all its receivers, a sender can identify receivers that have not received the last few messages sent by the sender. The sender can also identify if, due to a software error, one or more receivers have got a message that was never sent by the sender. Another advantage of using Session Messages is that the sender-side checks performed using Session Messages help to speed up the detection of errors in the experimental communication software.

Session Messages have been used in reliable multicast protocols like SRM [11]. As discussed in [11], one disadvantage is that as the number of nodes increases, the reception state becomes quite big. Very large groups pose overheads both in terms of the size of the Session Messages and the overall number of Session Messages exchanged between peers. Size of the reception state in Session Messages can be kept small by organizing the nodes hierarchically so that each node reports its reception state to only a small subgroup. This reduces the size of the Session Message as well as the number of members it has to be sent to. Another approach involves dynamically adjusting the generation rate of Session Messages in proportion to the group size as done in [11]. These schemes can save energy and bandwidth in large sensor networks.

Based on information in incoming Session Messages, the Inter-cSimplex Message Thread at the sender-side maintains the following statistics for each of its receivers:

- Number of times the receiver did not get the last few Data Messages sent out by the sender, after accounting for regular transmission delays.
- Number of times the receiver claimed to have received a message that the sender has not yet sent

In addition, the decision functions for both the Data and Inter-cSimplex Message Threads can be extended with checks specific to the application at hand.

D. Thresholds for Stability Checks

When the experimental module is executing, cSimplex compares the above error statistics against *thresholds* set for each type of error based on the performance of the safety module under the same environment. If any of the errors exceed the corresponding threshold, it denotes a likely software fault and cSimplex initiates a switch to the safety module. Thresholds are established by an initial *calibration run* that executes the safety module on all nodes for a sufficiently long period of time. The calibration run is done only once in a given environment, and after this new experimental modules can be inserted without shutting down the system.

Establishing thresholds serves two purposes. Firstly, it helps to account for the effects of environmental noises such as congestion and transient link failures, which affect both the safety and the experimental modules. Thus it is possible to distinguish between errors like message losses and delays due to environmental factors and those due to software faults in the experimental module. By repeating the calibration run in different environments, different sets of thresholds can be generated to model each of these environments. Secondly, thresholds represent the performance of the reliable safety module in a given environment. This forms a *performance floor* against which the performance of the experimental module can be compared. Thus cSimplex monitors the experimental software for two kinds of violations:

1. Safety Violations: Ensure that the system reliability under the experimental module is no worse than the reliability of the highly reliable safety module
2. Performance Violations: Ensure that the experimental module, which is intended to give higher performance, does indeed perform better the safety module

E. Statistical Methods

Several statistical approaches can be used to calculate thresholds. This section describes different approaches that were implemented to assess their accuracy, and the shortcomings of some of these approaches. We term the final approach, adopted after evaluating all the alternatives, as the *Multiple Period Method (MPM)*. We define *Statistics Period (P)* as the number of Data Messages sent or received, after which statistical measurements are made. All methods except MPM use a single Statistics Period.

1. Long-Term Error Rate

During the calibration run, counters for each error type are incremented each time a corresponding error is detected. Thresholds are calculated only at the *end* of the calibration run. However while running an experimental module, statistics cannot be compared against thresholds only at the end of the run since the aim is to detect errors in the software while it is executing and switch to the safety

module without stopping the system. Thus periodic checks are needed. Besides, the absolute number of errors cannot be compared, since the total number of messages exchanged differs across runs. Thresholds are therefore errors expressed as fractions of the total number of messages sent or received at the end of calibration. While running the experimental module, at the end of every P messages, the total number of errors since the beginning of the run is expressed as a fraction of total messages and compared against the corresponding threshold. If any of these error fractions exceeds its threshold, it signifies an error in the experimental module.

However, this approach tends to be too strict, since the system 'never forgets' past errors. Thus it has a higher incidence of *false alarms*, where a non-faulty communicator is terminated due to mistaken environmental noises.

2. Per-Period Error Rate

During the calibration run, counters for the various types of errors are reset at the end of every period, P. For each error type, the maximum counter value observed over all periods is taken as the final threshold. While running the experimental module, at the end of every P messages the errors noticed in that period are compared against the appropriate thresholds in order to detect a fault in the module, and the error counters are reset.

The principal shortcoming of this method is that it cannot detect a faulty communicator that introduces errors repeating at a low but constant rate. For example, if the threshold for losses is 2 messages per period, a faulty experimental module that repeatedly causes one message loss per period will go undetected. A shorter period would have been able to detect the error. The issue here is that a short statistics period ensures less latency in detection of errors, but it can also raise false alarms. A long statistics period ensures better statistics, but has higher detection latency.

3. Per-Period Error Rate with History

The problem with the previous method was that the system did not maintain a history of errors in previous periods. Thus in this method, the calibration run is similar to the Per-Period Error Rate method, but when running an experimental module, each error counter is now a sum of two parts, counter-old and counter-new. This moving window with an overlap provides a history of past errors. At the end of every P/2 messages, counter-old becomes equal to counter-new and counter-new is reset. Every time an error is noticed, counter-new is incremented. If (counter-old + counter-new) > threshold, the communicator is faulty. However this approach also cannot catch low-rate repeating errors. For example, say:

Loss threshold = loss of 2 messages per period.

Now, a faulty communication module causes one message loss in every alternate half-period (P/2) as shown in Fig. 5.

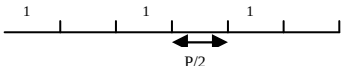

Fig. 5. Failure of Per-Period Error Rate with History

This faulty module will not be detected since the number of errors in any period, P, is one, which is below the threshold of two. However, a communication module that causes constant message losses is clearly faulty.

4. Multiple Period Method Used by cSimplex

As the name suggests, this method uses multiple statistics periods. For our experiments, three periods were heuristically chosen: P1=10, P2=50 and P3=100 messages, since these were determined to perform well without imposing too much computational overhead. Each error counter now is represented by a set of three counters: (c1, c2, c3). c1 is a short term counter. It is reset after every P1 messages. c2 is the medium term and c3 is the longer term counter. They are reset at the end of P2 and P3 messages respectively. The threshold for each type of error is also represented as a 3-tuple (T1, T2, T3). These are the maximum number of occurrences of that type of error over all periods of length P1, P2 and P3 respectively during the calibration run. Advantages of MPM over the other approaches are:

1. It combines the advantages of having a short period with those of a long period in the single period approach. If, due to a faulty communication module, there is a sudden burst of errors that exceeds the corresponding threshold T1, the communicator is terminated immediately without a long delay. At the same time, if the communicator causes periodic errors at a low frequency, the long-term threshold T3 catches the error even if it does not exceed the short-term threshold T1.
2. It is neither too strict nor too lenient in dealing with faulty communication modules. Thus the incidence of both false alarms and failure to detect an erroneous module are much lower, as seen from the experimental results. In methods that use a single statistics period, the length of the period is very important in determining the performance. However, in MPM, the threshold of the larger period detects faults missed by a higher and hence more lenient threshold for the smaller period. Thus it is noticeably superior to a single period approach.
3. Since error counters are compared against thresholds every time an error occurs and not just at the end of a period, errors can be detected very quickly. The detection latency or time to identify a faulty communication module is inversely proportional to the frequency of the error.

The only limitation of this method is that it may not be possible to detect errors in the experimental module that

repeat at a frequency greater than the longest statistics period, P3. However, this is a practical limitation since a period cannot me made infinitely long if it is to detect errors while the system is functioning, in order to allow online upgrades. Besides, as seen in the experimental results, this is usually not a serious limitation since the longest period can be made sufficiently large. The periods can be defined based on the characteristics of environmental noises and application behavior.

F. Distributed Architecture

Since cSimplex extends the Simplex Architecture to a distributed system, it must deal with issues like distributed startup, and distributed termination of faulty experimental modules. Management messages called *Startup Messages* are used to communicate with peers when a node first comes up. If cSimplex on a node detects that the experimental module violates the stability definition, it terminates the experimental communicator on that node and sends *Switch Messages* to active peers. It resends these messages to peers who do not acknowledge it through a *Switch ACK Message*. Once all acknowledgements are received, cSimplex on each node executes the safety communicator. If there are outstanding acknowledgements, possibly due to environmental factors like congestion or hardware failures, the node stops itself. This constitutes a *two-step approach to distributed termination* of faulty communication modules.

V. cSIMPLEX EXPERIMENTS

A. Experimental Setup

Our implementation setting consists of scattered acoustic sensors with embedded microcomputers, called MOTES [16], and three-wheeled robots controlled by on-board computers that act as mobile base stations for the MOTES. An example application scenario could be acoustic sound classification and localization in surveillance applications. The robot used is a Palm Pilot Powered Robot Kit (PPRK)[2]. The Palm Pilot that controlled the robot was replaced by a MOPS/520 embedded computer, based on the PC/104 CPU-board and running Linux. This was done to provide 802.11b wireless LAN connectivity for inter-robot communication, something that is not supported by the Palm. A robot with a base MOTE connected through the serial port serves as a base station for the scattered MOTES. cSimplex runs on each base station robot. GPS-equipped MOTES can communicate their position to the base station by radio transmissions to the base mote. Since the MOTES currently do not have GPS support, their

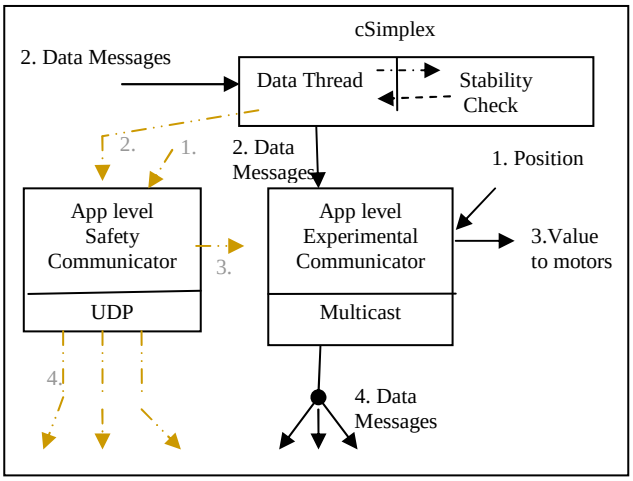

Fig. 6. Experimental Setup

positions are hard-wired for the purposes of these experiments. The robots obtain their position by contacting a vision server that tracks the robots using cameras and visual tracking software. The mission for the robots is to position themselves through mutual communication and co-operation to maximize sensor coverage and network connectivity.

The application-level communicator on each robot gets its position from the visual tracking server. As seen in Fig. 6, it periodically communicates this and other information to peer-robots. Based on the information from peers, each robot base station calculates the position it should move to so that global coverage of sensors is maximized. Communication is essential for the robots to position themselves optimally as well as to avoid collisions as they move. If the topology changes due to failure of some MOTES, the base stations can re-position themselves to maintain maximum network connectivity. Thus the network is self-healing.

The application-level safety communicator on each robot uses multiple point-to-point UDP transmissions to communicate with each of its peers. UDP is a well-tested protocol and hence the safety communicator together with the system-level UDP stack forms the reliable software backup. However, using individual UDP transmissions requires O (n^2) message exchanges between n robots, and this is repeated for every period. Thus the safety module is inefficient in terms of both energy and bandwidth. The more efficient, experimental communicator uses a new multicast group communication protocol that is being tested. All the robots in an area join a multicast group. By sending only one copy of the information to the network and letting the network intelligently replicate the packet only where needed, multicast conserves bandwidth and network resources at both the sending and receiving ends.

The experimental modules on the robots carry out communication and the safety modules do not come into function until required, as depicted by the dotted arrows in

[2] developed by the Robotics Institute at Carnegie Mellon University and sold by Acroname Inc, under a licensing agreement. http://www.acroname.com/robotics/info/PPRK/PPRK.html

Fig. 6. cSimplex on each robot performs statistical checks on the messages exchanged by the experimental software. It uses the stability definition and the user supplied application specific requirements to detect faults in the experimental module. e.g., a robot does not receive messages from one or more peers due to an error in either the application or system-level software. cSimplex then initiates a distributed switch to the UDP-based safety module on all robots. Thus both offline upgrade of the system protocol stack as well as online upgrade of the application-level communicator to a higher performance new version, can be done while guaranteeing that the baseline communication is always maintained.

B. Experimental Results

In order to measure the effectiveness of our approach, we performed tests by introducing various kinds of software faults in the experimental communication module. We measured the performance of the system as the percentage of faulty modules that it detected. We term this the *accuracy* of the system. The higher the accuracy, the better is the performance of the system. However, a system that is overly cautious and labels every module as faulty will derive very high performance scores with this measure. Hence, we complemented our performance measure with another measure for the occurrences of any false positives or *false alarms*. A false alarm occurs when cSimplex declares a module to be faulty, but in fact, it is not. This is often due to environmental noises that are mistaken for software errors. A false alarm is analogous to a Type-II error in statistics and is measured as the percentage of fault detections that are incorrect. The lower this measure is, the lower the number of errors related to false positives. Together, the accuracy and false alarm measures indicate the effectiveness of the system.

In our experiments, we measured the following metrics

$$\text{Accuracy} = \left(\frac{\#\text{ times faulty module was detected correctly}}{\#\text{ times faulty modules were introduced}} \right) \times 100$$

$$\text{False alarm} = \left(\frac{\#\text{ times the module terminated was not faulty}}{\#\text{ times cSimplex declared a fault}} \right) \times 100$$

$$\text{Right decisions} = \left(\frac{\#\text{ Right decisions made by cSimplex}}{\#\text{ experimental runs}} \right) \times 100$$

where a 'right' decision is both declaring a fault when the experimental module is faulty and not declaring a fault when the module is not faulty.

Each test in Table 2 studies one type of software error based on the common errors highlighted in Table 1. In all, 16 tests were performed, with each test consisting of 30 runs of the experimental module. In each run, it was randomly decided, with probability 0.5, whether or not to introduce a certain type of software fault in the experimental communicator on each node. If a faulty communicator was introduced on a node, it caused errors at a randomly generated rate. For software faults that cause message losses or delays, the test was repeated with environmental noises in the form of network congestion, created by introducing high external traffic. The following is a brief description of the software fault introduced in each test:

1. Software faults that cause messages to be sent to nodes that are not valid group members, e.g. due to errors in the group management module of the multicast protocol. This test studies the case where on introducing the experimental module, all nodes manifest the fault, but possibly at different rates.
2. Similar to Test 1, but only some nodes manifest the fault, i.e. random messages from only some senders are delivered to invalid recipients. E.g. this could be due to an erroneous multicast group join or leave call in the application-level communicator code that gets executed on some nodes, but not on others, since the nodes have different positions.
3. Software faults that cause a message sent to a multicast group to be delivered to only a subset of the members. This can occur due to:
   a. Faults in the multicast protocol, *e.g*. in the routing software that loses messages to some recipients.
   b. Routing software faults can cause packet storms that create excessive traffic and result in some members not receiving certain messages
   c. Local reception errors due to software faults on some recipient nodes.
4. Test 3 repeated with environmental noise.
5. Faults that cause a message sent to a multicast group to not be delivered to *any* of the intended recipients.
6. Test 5 repeated with environmental noise.
7. Faults causing messages to be delayed by random amounts that exceed application-acceptable delay limits. E.g. due to software faults in either the system or application-level modules, that cause packet storms or routing loops.
8. Test 7 repeated with environmental noise.
9. Studies performance in a harsh scenario with very large amounts of delays (5 to 6 times the application's tolerance) and with environmental noises. This is important for applications with a very low delay-tolerance.
10. The aim of this test is complementary to those of previous ones. *This test does not introduce any software faults*. Instead it introduces message delays within the acceptable range. Such small delays can occur due to regular operational factors, like transmission delays, varying execution times at different nodes due to execution of other, possibly higher priority tasks, or application delays, like lengthy data processing. The aim is to see if

cSimplex mistakes these to be software faults and raises false alarms

11. Test 10 repeated with environmental noise.

12 & 13. Software faults that introduce duplicate messages. *E.g*. when code is copy-pasted, variable names may accidentally be left unchanged, resulting in sending the same message again. In Test 12 all recipients receive duplicate messages, while in Test 13 only some do.

14. Faults that cause one or more valid group members to not receive *any* messages from a particular sender.

15. Test 14 repeated with environmental noise.

16. Software faults that cause a node to receive spurious messages that were never actually sent by the sender.

Table 2 presents the results of the 16 tests. It should be read as follows: E.g. In test 1, cSimplex had an accuracy of 100% (detected all faulty modules), 0% false alarms (never mistook a faultless module to be faulty) and 100% right decisions (all decisions made by cSimplex were correct).

TABLE 2
SUMMARY OF TEST RESULTS

#	Test	% Accuracy	% False Alarm	% Right Decision
1	Invalid Receiver, All Nodes	100	0	100
2	Invalid Receiver, Some Nodes	100	0	100
3	Loss to Subset of Receivers	100	0	100
4	Loss to Subset, With Noise	100	0	100
5	Loss to All Receivers	100	0	100
6	Loss to All, With Noise	96	4	93.33
7	Delay > Tolerance	100	0	100
8	Delay>Tolerance,With Noise	100	0	100
9	Delay>>Tolerance,With Noise	100	13.33	86.67
10	Small Delays (Not Error)	-NA-	-NA-	100
11	Non-error Delays, With Noise	-NA-	-NA-	100
12	Duplicates to All Receivers	100	0	100
13	Duplicates to Subset of Rcvrs	100	0	100
14	No Messages from a Sender	100	0	100
15	No Messages from a Sender, With Noise	100	0	100
16	Spurious Messages	100	0	100

TABLE 3
OVERALL METRICS

	Minimum	Maximum	Average
Accuracy	96 %	100 %	99.71 %
False Alarm	0 %	13.33 %	1.24 %
Right Decision	86.67 %	100 %	98.75 %

Results show that a combination of sender and receiver side stability checks along with the Multiple Period Method achieves low latency in detecting bursty errors, while also detecting errors that are widely distributed or that repeat at a low frequency. In Test 3, when the probability of error was very low and only one node had a faulty communicator, message losses occurred at a low frequency. In this case, it was seen that the error was detected by the medium and long term threshold, and not by the short term threshold. But with a high overall error rate, either due to multiple faulty nodes and/or high error rates on each off them, losses occurred in bursts and were quickly detected by the short term threshold. In Tests 14 and 15, when a node does not receive *any* messages from a particular sender, it does not detect a gap in sequence numbers received and hence the receiver cannot detect the error using timers. In these tests, the sender-side checks detect the fault, thus demonstrating the need for both sender and receiver side checks.

Results, as in Test 8, show that the accuracy is very high even when detecting software faults in a noisy environment. The software errors did not go undetected in spite of the high loss thresholds that had been set to account for environmental noises. At the same time, message losses due to congestion were not mistaken for software faults. Hence false alarms were not raised.

The minimum for accuracy (96%) occurred in Test 6, when higher error thresholds were set to account for the noisy environment. A faulty communicator went undetected in a few cases where software faults were introduced only on one node and the error rate at the node was very small. The node was also directly in the path of noise due to network congestion. Since most of the losses were due to noise and only a very small fraction was due to software faults, cSimplex could not distinguish between the two. This was also the reason for the false alarms observed in Test 6.

The minimum for percentage right decisions coincides with the highest percentage of false alarms. This occurred when message delays due to software faults were much higher than the application's tolerance and environmental noises were high (Test 9). In such a harsh environment, the accuracy was 100% while false alarms rose to 13.33 %. This also reflects the design decision that in a harsh setup, it is more important to guarantee safety of the system, even at the cost of occasional false alarms. In case of a false alarm, the mission is still accomplished by the lower performance safety module. Overall, it can be said that for most commonly occurring software faults in communication, the accuracy is very high and the percentage of false alarms quite low.

We also analyzed processing, memory and bandwidth overheads introduced by cSimplex, as these are important constraints in a sensor network. Results show that overheads were not significant for our sensor network application. However, the numbers are application-specific. Details are not presented here due to space constraints.

VI. CONCLUSIONS

We have developed an architecture to support reliable upgrade of the multicast-based group communication software in sensor networks despite residual errors in the new software. An important contribution of this work is the proposed definition of stability for multicast-based group communication. Another important feature is the two-level

system architecture, which allows reliable upgrade of both the system and application-level communication components. We demonstrated that the application-level communicator can be replaced online, without shutting down the system. Based on experimenting with different statistical methods, we propose the Multiple Period Method which performs well, as confirmed by experimental results.

The cSimplex architecture has been implemented from scratch and demonstrated to work in a system of robots acting as mobile base stations for a sensor network. Experimental results were evaluated based on three metrics that we have proposed to quantify the performance of the system, viz. accuracy, false alarms and right decisions. The experiments show that (Table 3):

- The accuracy of the system is at least 96 % and on an average cSimplex is 99.71% accurate.
- The percentage of false alarms is extremely low. It is always under 13.33%, and averages 1.24 %.
- The percentage of right decisions is always greater than 86.67 % with an average of 98.75 %.

Our approach ensures that if the experimental module is detected to be faulty or does not perform as expected, the mission is accomplished by the safety module, without any disruption in the functioning of the system.

An area for future work is to integrate cSimplex with static compiler checks to ensure memory safety of the system-level component of the experimental module. Another interesting approach is to implement cSimplex as a middleware component so that it can be made portable to different devices and allow multiple platforms to inter-operate. Since the cSimplex architecture makes it easy to plug in new stability checks, it can be extended to application domains beyond sensor networks. By implementing other applications based on the cSimplex architecture and incorporating ideas from them, cSimplex can ultimately be developed into a generic framework for reliably upgrading the communication protocol in a variety of systems.

Guaranteeing reliable upgrade of communication software in sensor networks is a real-world problem. The work presented here provides one of the first solutions to this problem in a structured manner. It also extends the Simplex architecture in two ways: by extending Simplex beyond control systems to reliable upgrade of group communication protocols and by adapting it to a distributed system with a simplex component on each node. The proposed architecture, which can be easily extended to other reliable upgrade problems, will facilitate a paradigm shift in system evolution from static design and extensive testing to reliable upgrades of critical communication components in networked systems, thus also enabling substantial savings in testing time and resources.

ACKNOWLEDGMENT

We thank Dr. Kumar and his team for providing us the visual tracking software. Thanks also to Travis Bullock for implementing the robot navigation code and Kihwal Lee for his assistance with hardware setup.

REFERENCES

[1] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, No. 4, July/August 2001, pp. 20-28.

[2] Juan-Mariano de Goyeneche, "*Multicast over TCP/IP HOWTO*," v1.0, March 1998, Chapter 9, pp. 21-22, http://cd1.ee.fhm.edu/AlleDateien/Multicast-HOWTO.PDF

[3] C. W. Wu, Y. C. Tay, and C. K. Toh, "Ad hoc Multicast Routing Protocol Utilizing Increasing id-numberS (AMRIS) functional specification", Internet-Draft, November 1998.

[4] E. Bommaiah, M. Liu, A. McAuley, and R. Talpade, "AMRoute: Ad hoc Multicast Routing protocol", Internet-Draft, August 1998.

[5] J. J. Garcia-Luna-Aceves and E. L. Madruga, "The Core-Assisted Mesh Protocol", *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 8, August 1999.

[6] P. Sinha, R. Sivakumar, and V. Bharghavan, "MCEDAR: Multicast Core-Extraction Distributed Ad hoc Routing," *Proc. IEEE Wireless Communications and Networking Conference*, 1999.

[7] M. Gerla, S.-J. Lee, and W. Su. "On-Demand Multicast Routing Protocol (ODMRP) for Ad-hoc Networks," Internet-Draft, Jan 2000.

[8] E. Royer, and C. E. Perkins "Multicast Operation of the Ad-hoc On-demand Distance Vector routing protocol," *Proc. 5th ACM/IEEE Annual Conf. on Mobile Computing and Networking*, Aug 1999.

[9] W. Heinzelman, J. Kulik, and H. Balakrishnan, "Adaptive protocols for information dissemination in wireless sensor networks," *Proc. 5th ACM/IEEE Mobicom Conference*, Seattle, WA, August 1999.

[10] K. Obraczka, "Multicast transport protocols: a survey and taxonomy," *IEEE Communications*, vol. 36, No. 1, January 1998.

[11] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Trans. on Networking, 1995*.

[12] Sape J. Mullender, "Fault-tolerant broadcast and related problems," *Distributed Systems*, chapter 5. Addison-Wesley, 1993.

[13] B.Bershad, S. Savage, P. Pardyak, E.G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the spin operating system," *Proc. 15th ACM Symposium on operating System Principles* (SOSP-15), 1995.

[14] K. Wehner, "An architecture for online multicast routing upgrade," Masters Thesis, University of Illinois at Urbana-Champaign, 2000.

[15] M. Gagliardi, "Simplex: A technology for rapid reliable upgrade," Carnegie Mellon Software Engineering Institute Symposium, Sept 1999.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System architecture directions for network sensors," ASPLOS-2000.

[17] L. Hatton, "N-version design versus one good design," IEEE Software, Vol. 14, No. 6, Nov/Dec 1997, pp. 71-76.

[18] J. P. Macker, Eric Klinker, and M. Scott Corson, "Reliable multicast data delivery for military networking," *Proc. IEEE MILCOM 96.*

[19] S. Han, K. G. Shin and H. A. Rosenberg, "DOCTOR: An integrated software fault injection environment for distributed real-time systems," *Int. Computer Performance and Dependability Symposium. (IPDS'95)*, pp. 204-13.

[20] "Configuring broadcast/multicast suppression," Catalyst 5000 Family Software Configuration Guide, © 1992-2001 Cisco Systems.

[21] P. Koopman, E. Tran, and G. Hendrey, "Toward middleware fault injection for automotive networks," *Fault Tolerant Computing Symposium*, June, 1998, pp. 78 - 79.

[22] S. Lim, K. Lee, and L. Sha, "Ensuring integrity and service availability in a web based control laboratory," *Journal of Parallel and Distributed System, Special Issue on Security in Mission Critical Real-time Systems*.

[23] L. Sha, "Dependable system upgrades," *Proc. IEEE Real Time System Symposium*, 1998.