Coordinated Broadcast-based Request-Reply and Group Management for Tightly-Coupled Wireless Systems

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

Abstract—**As the domain of cyber-physical systems continues to grow, an increasing number of tightly-coupled distributed applications will be implemented on top of wireless networking technologies. Some of these applications, including collaborative robotic teams, work in a coordinated fashion, whereby a distinguished node takes control decisions and sends commands to other nodes, which in turn perform the requested action/operation and send back a reply/acknowledgment. The implementation of such interactions via reliable point-to-point flows may lead to a significant performance degradation due to collisions, especially when the system operates close to the capacity of the communication channel. We propose a coordinated protocol which exploits the broadcast nature of the wireless medium in order to support this application-level interaction with a minimal number of message transmissions and predictable latency. The protocol also comes with group management functionality, allowing new processes to join and existing processes to leave the group in a controlled way. We evaluate a prototype implementation over WiFi, using a simulated setup as well as a physical testbed. Our results show that the proposed protocol can achieve significantly better performance compared to point-to- point approaches, and remains fully predictable and dependable even when operating close to the wireless channel capacity.**

I. Introduction

Many applications from the domain of cyber-physical systems [1] implement a distributed control logic over a wireless medium. A typical example include teams of unmanned vehicles (UVs) that perform collaborative missions e.g., the aerial scanning of an area of interest with a certain flight formation [2]. Such applications, require tightly-coupled reliable interactions between the UVs over the wireless medium. Moreover, the aspect of control poses the requirement of predictable system responsiveness, which translates to predictable network interaction latencies.

A popular implementation approach for these interactions is to adopt a coordinated design in the spirit of a group RPC [3] whereby a distinguished node takes control decisions and sends commands to other nodes, which in turn perform or schedule the requested action/operation and send back a reply/acknowledgment. Conventionally, such a communication pattern can be supported on top of multiple reliable point-to-point flows such as those provided from TCP. However this may lead to increased network traffic and higher latency. Furthermore, if performed in an un-coordinated way, it can lead to increased contention for the shared medium, resulting in unpredictable latencies. Both problems are particularly undesirable in a wireless setting, since they can lead to a significant performance degradation.

Taking a different approach, we propose a coordinated 1-N request-reply protocol, which achieves reliability, high throughput and predictable low latency. This is achieved by exploiting the broadcast nature of wireless communication in conjunction with a simple but effective scheme for avoiding contention during the transmission of replies back to the coordinator. The protocol also comes with integrated group management functions, allowing processes to join and leave the group at any point in time. Notably, our protocol can serve as a foundation for higher-level system support, such as the tightly-coupled UAV team programming approach presented in [4].

The main contributions of the paper are: (i) we present a coordinated protocol that supports request-reply interaction in dynamically changing wireless process groups; (ii) we describe the protocol in detail, including a high-level algorithmic description and an informal proof for the key group management properties; (iii) we evaluate an implementation of our protocol for a 802.11 WiFi network, using a simulated setup and a wireless testbed; (iv) we compare our protocol with TCP-based and simpler unicast-based approaches, showing that it achieves significantly better and predictable performance, especially when operating close to the channel capacity, for WiFi that favors unicast messaging. It is important to note that the proposed protocol is completely agnostic of the underlying network technology, and will work very well also on top of much simpler wireless technologies that do not feature an advanced medium-access-control (MAC) mechanism.

The rest of the paper is structured as follows. Section II introduces the application-level interface for a coordinated request-reply interaction within a process group. The protocol for supporting this API on top of a distributed system with broadcasting capability is described in detail in Section III. Section IV discusses the key performance properties and semantics of the supported request-reply interaction, while Section V gives informal proofs for the most important group management properties of the protocol. Section VI presents a performance evaluation and comparison with different point-

TABLE I: Application-level primitives

Init($pid, pids$)**:** Initializes the protocol with identifier pid for the local process. If $pids$ is empty, the local process must explicitly join the group via the $Join$ primitive. Else, $pids$ contains the members of the group, including pid.
RequestReply($pids, data, procT$)**:** Sends an application-level request to the set of processes $pids$ with payload $data$ and an estimated processing delay $procT$, and returns the set of replies produced.
ProcessRequest($data$)**:** Handle the next application-level request with payload $data$, and return the reply for it.
Join()**:** Join the group.
Leave()**:** Leave the group.
CheckJoins($joinT$)**:** Wait $jointT$ for processes to join.
CheckLeaves($leaveT$)**:** Wait $leaveT$ for processes to leave.
BecameCoord($pids$)**:** The local process has coordinator status, for the group view which includes processes $pids$.
GroupChanged($pids$)**:** The current group view changed (due to a join/leave/failure), and includes processes $pids$.

to-point approaches. Section VII provides an overview of related work. Finally, Section VIII concludes the paper and points to directions for future work.

II. Application interface

We target distributed process groups, where a distinguished process acts as a *coordinator* and issues requests to one or more *ordinary* processes (nodes). In turn, the addressed ordinary processes perform the requested action or operation, and send back a reply. The reply serves as an application-level acknowledgment but may also contain additional information. Furthermore, new processes may join the group, and existing group members can leave the group, anytime. Processes that fail are also removed from the group. In case the coordinator fails or leaves the group, one of the ordinary processes takes over as the new coordinator.

This functionality can be exposed to the application via the primitives of Table I. The $RequestReply$ call and $ProcessRequest$ event embody the basic application-level request-reply interaction. The $Join$ and $Leave$ primitives let a process join and leave the group, respectively; it is up to the coordinator to check for and handle such processes, via $CheckJoins$ and $CheckLeaves$. Finally, the $BecameCoord$ event informs the application that it took over as the new coordinator, while $GroupChanged$ notifies it about any subsequent group changes.

The motivation for introducing the $CheckJoins$ and $CheckLeaves$ primitives is to give the application more flexibility but also more control on the network traffic. For instance, the coordinator might let processes join or leave the group under certain application-specific conditions to which the underlying protocol is oblivious. Then, it is pointless (and costly) to keep the respective group management protocol functions active at all times. Also, some applications might be able to work without such group management functionality, in which case the corresponding protocol components could be dropped rather than be carried along as dead code; of course, the group could still change due to process failures.

Note that only the process that becomes the coordinator is notified accordingly, in order perform any application-level recovery actions, if needed, and resume the group's operation. Also, only the coordinator is notified about group changes so that it can adjust its operation. Note that the coordinator can easily forward such notifications to all other group members, under the control of the application, via the $RequestReply$ primitive. In a similar vein, it is up to the application to use this primitive in order to back-up critical coordinator state to other members, as needed.

III. The GCBRR protocol

To support the above API we propose a 1-N request-reply and group management protocol, which exploits the broadcast nature of the wireless medium to eliminate superfluous messages and avoid network contention. We refer to this protocol as the *Coordinated Broadcast-based Group Request-Reply* protocol (or GCBRR).

A. System model and assumptions

Let there be N processes (nodes), with monotonically increasing identifiers, p_i, $1 \leq i \leq N$. The processes share the same broadcast domain and communicate via broadcast or unicast transmissions. Both types of transmissions are unreliable. Broadcasts are non-atomic and might arrive at only a subset of the processes. We assume bounded messaging delay: a message will arrive at its destination(s) within $MsgT$ after it was sent, or not at all.

Processes may fail at any point in time. We assume synchronous and reliable fault detection: if p_i fails then all correct processes will be notified about this failure within $FaultDetectT$. In case of multiple failures, the respective notifications may reach the correct processes in a different order. But a failure notification for p_i cannot overtake the last message that was sent by p_i. Also, we assume that once a process leaves the group, it behaves like a failed process (in practice, this can be achieved by having the process explicitly reject messages coming from the group).

Finally, a process that has failed or left the group might wish to join the group, again, at a later point in time. We assume that processes do not attempt to re-join earlier than $FaultDetect$ after they failed or left the group. This ensures that the messages sent by a process as part of the join procedure do not overtake the notification about its previous departure/failure. Alternatively, one can rely on process incarnation numbers.

B. Protocol structure

The GCBRR protocol is structured in a modular way. It consists of the: (i) the basic *application request-reply* protocol; (ii) the *join* protocol; (iii) the *leave* protocol; (iv) the *view-push* protocol; (v) the *fault handling and election* procedure. This makes it easier not only to describe but also to implement/test the protocol. Of course, these components are not entirely orthogonal to each other. But the interplay between them is limited and clearly defined. Also, it is straightforward to drop some of these components, if the corresponding functions are

TABLE II: Per-process protocol state

$state$	Membership state
$myid$	Identifier of the local process
$coordid$	Identifier of the coordinator process
grp	Group view including ticket information
$seqno$	Sequence number of last request sent/received
$rmask$	Reply bitmask for the current request
$replies$	Replies received from (ordinary) processes
$reply$	Reply of local (ordinary) process

TABLE III: Protocol messages

AREQ[$pid, k, rmask, data$**]:** Application request from pid (coord) with sequence number k for processes in $rmask$
ARPL[$pid, k, data$**]:** Application reply from pid for the request with sequence number k
JPOLL[pid**]:** Join poll from pid (coord)
JOIN[[pid**]:** Join request from pid, after a join poll
LPOLL[]: Leave poll (from coord)
LEFT[pid**]:** Leave notification from pid, after a leave poll
VPUSH[$pid, grpc$**]:** View-push request from pid (coord) for the new/updated group view $grpc$
VACK[pid**]:** View-push acknowledgment from pid

not required by the application. In the following, we discuss the individual components of GCBRR. Algorithms 1, 2, 3, 4, 5 and 6 provide corresponding descriptions in the form of high-level code.

C. Protocol state and messages

The protocol state maintained locally by each process is summarized in Table II. The messages types used in GCBRR for each of its components are shown in Table III.

The membership status of the local process is kept in variable $state$, and equals: $NORMAL$ if the process is a member of the group; $JOIN$ if the process is attempting to join the group (but is not yet a member of the group); $LEAVE$ if the process is attempting to leave the group (but is still a member of the group); $LEFT$ if the process has left the group.

The current local group view, which contains the identifiers of the process that are considered to be members of the group, is kept in the grp data structure. Importantly, grp includes per-process ticket number information. The process with the smallest ticket number is the coordinator; its identifier is stored in $coordid$. The assignment of ticket numbers to newly joining processes is done by the coordinator. The group view also contains information in order to differentiate between old and newly added members.

The sequence number of the last request sent/received (depending on whether the process is the coordinator or an ordinary process) is stored in $seqno$. The reply bitmask $rmask$ encodes the (ordinary) processes addressed in the current request, which should send back a reply: if p_i should handle the request then the ith bit of $rmask$ is 1 (else 0). The reply bitmap also defines the order in which these processes should reply: if the ith bit is the kth non-zero bit of $rmask$ then p_i must send its reply in the kth "transmission slot". Ordinary processes store in $reply$ their reply to the last request received; this may have to be re-transmitted, if requested by the coordinator. The $replies$ structure is used by the coordinator to keep the replies that arrive from the ordinary processes.

D. Basic request-reply protocol

When a new request-reply interaction starts, the sender/coordinator increases $seqno$, initializes $rmask$ to address the target processes, broadcasts a request message, and waits for the replies to arrive or a timer to expire. The timeout is set as a function of $MsgT$, the number of processes that are expected to reply, and the application request handling delay. Each reply that is received is added to the $replies$ structure, and $rmask$ is adjusted accordingly. When the timer fires and some processes have not replied, the request is re-transmitted. This is repeated until all target processes either reply or fail. The coordinator proceeds with the next application request once the previous one has been handled to completion.

When a process receives a request, it checks $rmask$ to determine whether it is among the addressed processes. If so, it compares the sequence number k with that of the last locally handled request $seqno$. If the request is new, it is handed over to the application, and the produce reply is stored in $reply$; else, the reply is already stored in $reply$. Replies are broadcasted, and as a result are received by other processes. The local process sends its reply following the order specified via $rmask$: if it is first in order, it does so as soon as the application produces the reply, else, right after it overhears the reply of the preceding process. To deal with message loss, a timer is set to expire in order to send the reply at the corresponding transmission slot.

E. Join protocol

The coordinator invites non-member processes to to join the group, by broadcasting a join poll message. It then waits a certain amount of time for corresponding requests. Each process that responds is added to the local group view, and is assigned the next ticket number.

When the waiting time is over, the coordinator propagates the updated view to the group, via the view-push protocol (described in the sequel). In a first phase, the new view is disseminated to the old members (without the ones that are currently joining the group). In a second phase, the view is sent to each of the newly joining processes, in increasing ticket order.

A process wishing to join the group enters the $JOIN$ state and waits for the coordinator to start a join poll. It then sends a join request with its identifier (this is a unicast message). The process remains in this state until it receives a group view from the coordinator. In the meantime, it might receive additional join poll requests, to which it responds as above. Processes that are already members of the group ignore join polls.

F. Leave protocol

Similarly to the join protocol, the coordinator invites group members to declare their intention to leave, via a broadcast, and waits some time for any responses.

If an ordinary process wishes to leave, it enters the $LEAVE$ state, and waits for the coordinator to issue a leave poll (as long as the process remains in the $LEAVE$ state, it is expected to continue handling incoming requests as usual). It then announces its departure via a broadcast, and enters the $LEFT$ state. This state transition is unilateral, without any further communication. If the coordinator wishes to leave the group, it does the same (without having to wait for a leave poll).

When a process receives a departure announcement, it follows the same procedure as for process failures (explained in the sequel). Note that, since such announcements can be lost, the leave protocol is a merely *best-effort* attempt to accelerate the removal of a process from the group view. If a departed process remains in the view of a member process, eventually, that process will receive a notification from the fault-detector.

G. View-push protocol

The goal of a view-push protocol is to commit a group view to a specified set of processes. It works along the lines of the request-reply protocol. In this particular case, the request carries the (updated) group view, and the target processes integrate it with their own local views, and send back an acknowledgment.

H. Fault handling & coordinator election

When a process is notified about the failure (or departure) of p, it removes p from its local group view. If p is an ordinary process, the coordinator stops waiting for replies from it. If p is the coordinator, the group member with the next smallest ticket number is elected as the new coordinator — without any further communication among the remaining group members. Before resuming normal operation, the new coordinator performs a view-push, if its view contains new members.

Note that the new coordinator can already be in the $LEAVE$ state. In this case, instead of taking over, it leaves the group following the leave protocol (see above), which in turn will trigger another election.

IV. Performance properties and semantics

The "critical path" of GCBRR in terms of performance is the basic request-reply protocol, which is designed to achieve several important properties: (i) It is contention-free, avoiding concurrent transmissions from different processes. (ii) It incurs the minimum number of transmissions needed for a 1-to-N request-reply interaction: the request and each reply are transmitted just once (assuming no loss). (iii) It achieves minimal latency since ordinary processes send their replies as fast as possible, according to the schedule specified by the coordinator. (iv) It is offers very predictable performance, and allows the system to operate close to the channel capacity. (v) It enables robust message recovery in case of loss, by addressing only the processes that have not responded. (vi) The values for the message transmission delay and application-level request processing delay can be chosen generously, and affect performance only in case of message loss. (vii) Last but not least, no assumptions are made about the under-

Algorithm 1 Protocol initialization

```
1: function INIT($id, pids$)
2:     $myid \leftarrow id$
3:     $seqno \leftarrow 0$
4:     $grp \leftarrow$ initGroupMembers($pids$)
5:     if $pids = \varnothing$ then
6:         $state \leftarrow LEFT$
7:         $coordid \leftarrow 0$
8:     else
9:         $state \leftarrow NORMAL$
10:        $coordid \leftarrow$ minTicket($grp$)
11:        if $myid = coordid$ then
12:            BECAMECOORD(procs($grp$))
13:        end if
14:    end if
15: end function
```

Algorithm 2 Basic request-reply protocol

```
1: function REQUESTREPLY($dsts, data, procT$)
2:     $replies \leftarrow \varnothing$
3:     $seqno \leftarrow seqno + 1$
4:     $rmask \leftarrow$ setBits($dsts \ominus myid$)
5:     while $rmask \neq 0$ do
6:         broadcast($AREQ[myid, seqno, rmask, data]$)
7:         $wt \leftarrow MsgT + procT +$ bitsSet($rmask$)$*MsgT$
8:         await($rmask = 0, wt$)
9:     end while
10:    return ($replies$)
11: end function

12: function OnRecv($AREQ[pid, k, rmask, data]$)
13:    if isBitSet($rmask, myid$) then
14:        if $k \neq seqno$ then
15:            $seqno \leftarrow k$
16:            $reply \leftarrow$ PROCESSREQUEST($data$)
17:        end if
18:        $myturn \leftarrow$ false
19:        $wt \leftarrow$ posBit($rmask, myid$)$*MsgT$
20:        await($myturn =$ true$, wt$)
21:        broadcast($ARPL[myid, seqno, reply]$)
22:    end if
23: end function

24: function OnRecv($ARPL[pid, k, data]$)
25:    if $k = seqno$ then
26:        if $myid = coordid$ then
27:            $replies \leftarrow replies \oplus (pid, data)$
28:            $rmask \leftarrow$ clearBit($rmask, pid$)
29:        else
30:            $myturn \leftarrow$ nxtBit($rmask, pid, myid$)
31:        end if
32:    end if
33: end function
```

Algorithm 3 Join protocol

```
function JOIN
    $state \leftarrow JOIN$
    await($state = NORMAL$)
end function

function CHECKJOINS($joinT$)
    broadcast($JPOLL[myid]$)
    $wt \leftarrow 2 * MsgT + joinT$
    sleep($tw$)
    if new($grp$) $\neq \varnothing$ then
        ViewPush(old($grp$))
        for each $p \in$ sort(new($grp$)) do
            ViewPush($\{p\}$)
        end for
    end if
end function

function OnRecv($JPOLL[pid]$)
    if $state = JOIN$ then
        unicast($pid, JOIN[myid]$)
    end if
end function

function OnRecv($JOIN[pid]$)
    $t \leftarrow$ nxtTicket($grp$)
    $grp \leftarrow grp \oplus (pid, t)$
    GROUPUPDATED($grp$)
end function
```

Algorithm 4 Leave protocol

```
function LEAVE
    if $mypid = coordid$ then
        $state \leftarrow LEFT$
        broadcast($LEFT[myid]$)
    else
        $state \leftarrow LEAVE$
        await($state = LEFT$)
    end if
end function

function CHECKLEAVES($leaveT$)
    broadcast($LPOLL[]$)
    $wt \leftarrow 2 * MsgT + leaveT$
    sleep($wt$)
end function

function OnRecv($LPOLL[pid]$)
    if $state = LEAVE$ then
        $state \leftarrow LEFT$
        broadcast($LEFT[pid]$)
    end if
end function

function OnRecv($LEFT[pid]$)
    OnProcessFailure($pid$)
end function
```

Algorithm 5 View-push protocol

```
function ViewPush($pids$)
    $rmask \leftarrow$ setBits($pids \ominus ypid$)
    while $rmask \neq 0$ do
        broadcast($VPUSH[mypid, rmask, grp]$)
        $wt \leftarrow MsgT+$ bitsSet($rmask$)$*MsgT$
        await($rmask = 0, wt$)
    end while
end function

function OnRecv($VPUSH[pid, grpc]$)
    if isBitSet($rmask, myid$) then
        if $pid \neq coordid$ then
            $coordid \leftarrow pid$
            $seqno \leftarrow 0$
            $state \leftarrow NORMAL$
        end if
        for each $(p, t) \in grpc$ do
            $grp \leftarrow grp \oplus (p, t)$
        end for
        $myturn \leftarrow$ false
        $wt \leftarrow$ posBit($rmask, myid$)$*MsgT$
        await($myturn =$ true$, wt$)
        broadcast($VACK[mypid]$)
    end if
end function

function OnRecv($VACK[pid]$)
    if $myid = coordid$ then
        $rmask \leftarrow$ clearBit($rmask, pid$)
    else
        $myturn \leftarrow$ nxtBit($rmask, pid, myid$)
    end if
end function
```

Algorithm 6 Fault handling and coordinator election

```
function OnProcessFailure($pid$)
    $grp \leftarrow grp \ominus (pid, *)$
    if $myid = coordid$ then
        $rmask \leftarrow$ clearBit($rmask, pid$)
        GROUPUPDATED($grp$)
    else if $pid = coordid$ then
        $coordid \leftarrow$ minTicket($grp$)
        if $myid = coordid$ then
            if $state = LEAVE$ then
                LEAVE
            else
                BECAMECOORD($grp$)
                if new($grp$) $\neq \varnothing$ then
                    ViewPush(old($grp$))
                    for each $p \in$ sort(new($grp$)) do
                        ViewPush($\{p\}$)
                    end for
                end if
            end if
        end if
    end if
end function
```

lying medium-access-control (MAC) mechanism. Therefore GCBRR can work very well even on top of simple radios that do not have an intelligent MAC.

Admittedly, the join and leave protocols are subject to contention, if several processes wish to join and respectively leave the group at the same time. We consider this to be a rare occasion for most applications, and even then a simple back-off mechanism will greatly reduce the probability of collisions. Note, however, that the join/leave protocols do not interfere with the basic request-reply protocol; it is up to the coordinator to activate them when desired, via the respective primitives.

In terms of application-level semantics, the request-reply interaction is synchronous, allowing the coordinator to drive the group in a tightly-coupled way, enforcing progress at all (addressed) processes at the same pace while also keeping track of their liveness. It is "exactly once" for processes that do not fail, and "at most once" for processes that fail. The application learns about such failures via the respective notifications; it can also infer a process failure by inspecting the returned replies. Notably, the interaction is "non-atomic", since a newly elected coordinator does not attempt to complete the last request-reply interaction that was initiated by the old coordinator. Atomicity does not seem to make sense here, because it is unclear how the new coordinator could handle, in a meaningful way, the replies for a request that was not issued by it. Of course, when the application is notified that it takes over as the new coordinator, it can perform any corrective actions it deems necessary.

V. Group management properties

The GCBRR protocol is designed for a group that has at least one correct process as a member at all times. If at some point all group members fail or leave, the group disappears and no longer exists. We also note that the (best-effort) leave protocol does not affect the essence of the group management functionality, and in the following consider process departures only due to failures.

Assume that group views contain entries (p, t) where p is a process identifier and t the ticket assigned to it. It is easy to show that for any two correct group members p_1 and p_2 with local views grp_1 and grp_2, it holds that $(p, t_1) \in grp_1 \wedge (p, t_2) \in grp_2 \implies t_1 = t_2$. In other words, all members adopt the same ticket assignment (STA). Moreover, three additional important properties hold: (i) At any point in time, only one correct process p can be elected as the coordinator (GM1). (ii) There is no deadlock, where the group is non-empty but it is not possible to elect a correct member process as the coordinator (GM2). (iii) The confirmation to a joining process that it became a member of the group is binding (GM3). Below we provide an informal proof sketch.

A. Uniqueness of the coordinator (GM1)

From STA it directly follows that $grp_1 = grp_2 \implies \mathit{minTicket}(grp_1) = \mathit{minTicket}(grp_2)$. As a consequence GM1 holds for any two processes with the same view. However, during the transition periods caused by group dynamics, some group members might have different views. These cases are discussed below.

Let $(p, t) \in grp_1 \wedge (p, t) \notin grp_2$. If p has failed, this does not affect GM1. It suffices to focus on views that differ only due to new processed joining the group.

Starting from a state where all group members have the same local view, assume that process $p_i, 1 \leq i \leq k$ join the group, and that the coordinator starts to push the new view that includes the respective entries (p_i, t_i). Let the view push occur partially, reaching p_1 but not p_2, so that $(p_i, t_i) \in grp_1$ and $(p_i, t_i) \notin grp_2$. Since the view is first pushed to the old group members, p_1 cannot be a new member if p_2 is an old member. If both p_1 and p_2 are old members, due to the ticket assignment scheme it holds that $t_1, t_2 < t_i$, so the newly joined processes p_i cannot affect the outcome of an election at p_1 or p_2. If p_2 is a new member, it is still in the JOIN state and cannot perform a coordinator election in the first place. Thus GM1 also holds during a group update phase.

B. No deadlock (GM2)

Given that GM1 holds, GM2 holds too, provided that if some correct process p_2 elects as the coordinator another correct process p_1 then p_1 is certain to also consider itself as a member of the group. This is trivially so if p_1 is already a member when p_2 joins the group. It is somewhat less obvious if p_1 and p_2 join at the same time (respond to the same join poll). In this case, GM2 holds because the coordinator pushes the updated group view to each new member in increasing ticket order.

To see why, assume that the coordinator pushes the view grp_c that includes entries for the joining members (p_1, t_1) and (p_1, t_2), with $t_1 < t_2$. Assume that the coordinator starts pushing the view in random order, but fails before completion. Also assume that all other old group members (which may have received grp_c) fail too. Finally, assume that p_2 has received grp_c and considers itself as a group member, but p_1 has not. Then, p_2 will elect p_1 as the coordinator. However, p_1 remains in the $JOIN$ state, waiting for the next coordinator to initiate the next join poll, leading to a deadlock.

C. Join commitment (GM3)

In order for GM3 to hold, it must be shown that once a joining process p receives grp_c with an entry (p, t) for it, p will never have to fall-back to the $JOIN$ state. Equivalently, it must be shown that if p receives grp_c, every other process p_i with $(p_i, t_i) \in grp_c \wedge t_i < t$, which could take over as a coordinator while p is still alive, has already received grp_c. Indeed this holds due to the order in which the coordinator performs a view-push.

Also, p will not be addressed in the basic request-reply protocol, unless it considers itself as a member of the group (its state is $NORMAL$). This is guaranteed because the coordinator can proceed with the next request-reply interaction only after having successfully completed the corresponding view-push. It is also for this reason that a view-push is performed when a new coordinator takes over: to ensure that new members are aware of their membership status. This is not required if the view does not include any new group members.

VI. Evaluation

We have implemented GCBRR for a Linux-based platform, as a user-space library that resides on top of the operating system. Our implementation uses RAW sockets (bypassing the IP stack). The library offers a set of primitives in the spirit of the API in Table I. This section presents a performance evaluation over WiFi, and discusses the most important results.

A. *Experimental setup*

We test our implementation using two different setups, a simulated 802.11 network and a 802.11 testbed.

Simulated setup: For the simulated setup we use the OpenNet simulator [5]. This is an open-source simulation environment, which is build on top of Mininet [6] and ns-3, combining their features to provide a realistic virtual wireless testbed infrastructure on a single PC. OpenNet uses Mininet to create a network of virtual Linux hosts, which can run conventional applications without any modification. Each virtual host features a virtual network interface which is internally connected via a TAP device to an ns-3 node. In turn, ns-3 provides the modeling of the 802.11 wireless channel between virtual hosts. To reflect the testbed setup, we configure ns-3 for the 802.11g protocol with a transmission rate of 1Mbps for both unicasts and broadcasts.

Testbed setup: For the real measurements we use the NITOS testbed [7] featuring ICARUS [8] wireless nodes. These are PC-class machines with an Intel Core i7-2600 Processor with 4 GB RAM and wireless Atheros 802.11a/b/g/n cards, running Linux. The testbed environment is protected from external traffic/interference. The nodes are placed in a grid, and can communicate with each other in 1 hop. The WiFi interface is accessed through the standard socket interface via *ath9k* network driver, and is configured to operate with the 802.11g protocol in ad-hoc mode. To make a fair comparison among broadcast-based and unicast-based approaches, we fix the transmission rate to 1 Mbps for both. It would also be possible to transmit broadcast packets at the highest rate of the wireless interface by using the pseudo-broadcast transmission approach presented in [9], but this would not contribute to the essence of our evaluation (the main point is for broadcasts and unicasts to be equally fast). The average round-trip time between two nodes is about 26 milliseconds for packets with a payload of 1500 bytes.

The simulated setup is used to perform experiments across a wide range of parameters, without requiring physical nodes. We then use the testbed to test selected scenarios in order to verify the trends we observed via simulations. In both cases, each process (group member) is placed on a different virtual/physical node.

B. *Basic 1-N request-reply*

To evaluate the raw performance of the basic request-reply interaction of GCBRR, we use an application that issues dummy requests and produces dummy replies, without performing any processing. Request and reply messages are filled with dummy data, so all transmitted packets have the maximum payload size (1500 bytes).

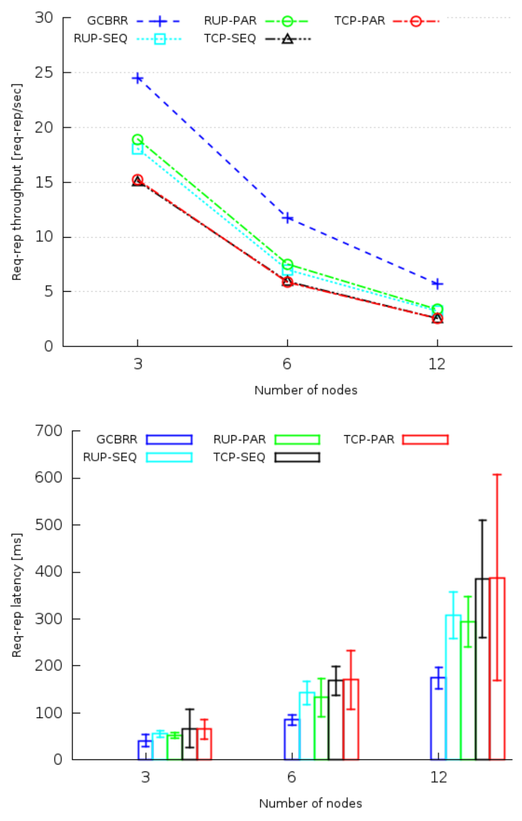


Fig. 1: 1-to-N request-reply throughput (top), and latency (bottom), for different group sizes.

We compare GCBRR with four reliable point-to-point transports: TCP-SEQ, TCP-PAR, RUP-SEQ and RUP-PAR (RUP stands for reliable unicast protocol). In the TCP variants, the coordinator keeps a separate connection with each ordinary process, which is reused for all interactions. RUP variants use plain datagrams over RAW sockets, with a simple acknowledgment scheme for the re-transmission of requests. SEQ variants perform the request-reply interaction sequentially, one process at a time; this avoids contention but comes at increased latency. In the PAR variants, the coordinator first sends the request to all ordinary processes, and then waits for the replies; this can reduce the total latency but also introduces more contention.

We perform experiments for groups of different sizes: 3, 6 and 12 nodes. To stress the network, we let the coordinator issue 1000 request-reply interactions at full speed, and record the total throughput and latency of each interaction. The results are shown in Figure 1.

As can be seen, GCBRR performs significantly better than all other variants, and the difference increases with the group size. For 3 nodes, GCBRR achieves x1.62 and x1.35 the throughput of TCP and RUP variants, going up to x2.22 and respectively x1.76 for 12 nodes. This is because GCBRR transmits fewer packets and scales better for a larger number of nodes. Note that for each unicast WiFi sends a MAC-level acknowledgement, which further increases the overhead of point-to-point variants. TCP achieves lower throughput than RUP due to the extra (TCP-level) acknowledgments, yet this difference becomes less pronounced for larger group sizes. Message loss is negligible in all cases, as the WiFi flow-control mechanism kicks in when the network is stressed.

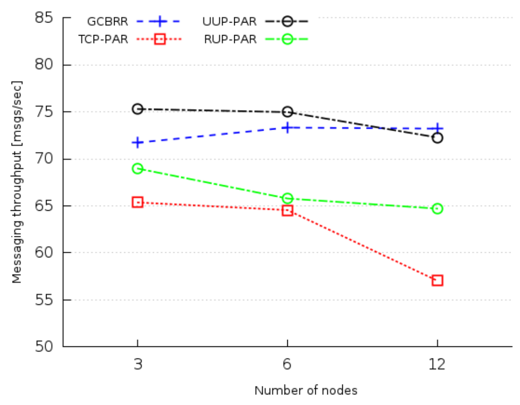
The difference in the number of packet transmissions also reflects on the latency of the request-reply interaction. Again, GCBRR outperforms all other variants, especially in larger groups. For 12 nodes, its latency is only about 1/2 and 2/3 that of TCP and RUP, respectively. RUP variants are generally better than TCP. Note that RUP-PAR has a slightly lower latency than RUP-SEQ because of the concurrent transmission of replies back to the coordinator. This advantage does not show in the TCP variants, due to their higher network traffic. In fact, TCP-PAR has a much larger variance than TCP-SEQ. This is an effect of the increased contention: although there is no message loss, WiFi slows down transmissions internally.

C. *One-way N-1 messages*

We also evaluate GCBRR regarding its ability to support a reverse, one-way message flow, from ordinary processes to the coordinator. In this case, the coordinator polls ordinary processes via an empty request, and receives their messages as replies. For comparison, we use TCP-PAR and RUP-PAR, where each ordinary process sends its messages to the coordinator over a TCP connection and a RAW socket, respectively. In RUP-PAR, messages are acknowledged explicitly following an alternating bit scheme. As an additional reference, we use the simplest possible best-effort approach where messages are sent via unreliable unicasts over a RAW socket, referred to as unreliable unicast protocol (UUP-PAR).

We measure the maximum message throughput that can be achieved in a group of 3, 6 and 12 nodes, at full speed. In GCBRR, the coordinator polls the ordinary processes 1000 times, while in TCP-PAR, RUP-PAR and UUP-PAR we let each process send 1000 messages to the coordinator. The aggregate throughput is calculated at the coordinator, by recording the time between the arrival of the first and the last message. At each ordinary process, we record the time that elapses between two consecutive message transmissions, which reflects the messaging delay. The results are shown in Figure 2.

Despite the polling overhead, GCBRR achieves higher message throughput than TCP-PAR and RUP-PAR, even for smaller groups. The difference increases for larger groups, where TCP-PAR and RUP-PAR lead to increased contention, and the relative cost of polling decreases. At 12 nodes, the throughput of GCBRR is x1.28 and x1.13 that of TCP-PAR and RUP-PAR, respectively. In this case, GCBRR even outperforms UUP-PAR which also has a very significant message loss of about 7%. Furthermore, unlike the RUP and TCP variants, GCBRR has a stable and predictable messaging delay.



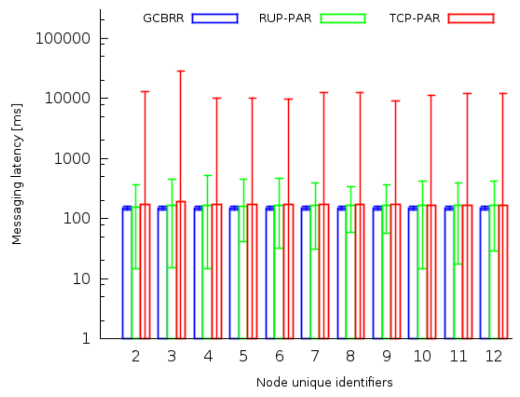


Fig. 2: Aggregate N-to-1 message throughput for different group sizes (top), and per-node messaging delay for an indicative run with 12 nodes (bottom; y-axis in log scale).

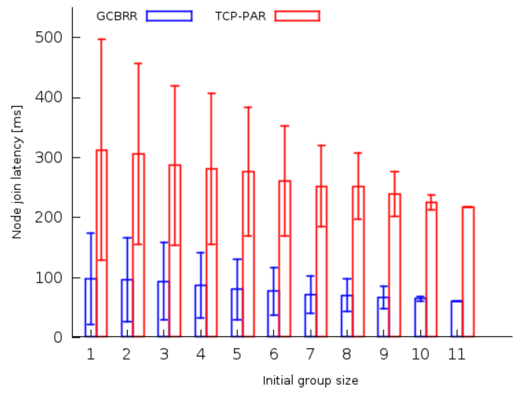
In fact, with TCP-PAR all nodes occasionally experience huge delays (the scale of the y-axis in the plot is logarithmic). Due to contention, TCP-PAR has some message loss, which leads to retransmissions and the activation of the TCP flow-control mechanism. In contrast, GCBRR and RUP-PAR did not need to perform any retransmissions. Also note that in TCP-PAR message sending sometimes appears to be instantaneous due to internal buffering.

We wish to note that GCBRR, due to its polling approach, is not suitable for sporadic messaging from ordinary processes to the coordinator. In such scenarios, where the contention on the network is also low, it is better to use an uncoordinated approach.

D. *Group management*

Finally, we evaluate GCBRR in terms of the group management overhead.

As a reference, we use a TCP-based approach where the coordinator keeps an open connection with each member process. A joining process opens a TCP connection to the coordinator when it receives a join poll (without sending a join request). It considers itself as a member once it successfully receives the group view over the connection. To leave the group, a process simply closes the connection to the coordinator (it does not send a leave announcement). If the coordinator wishes to



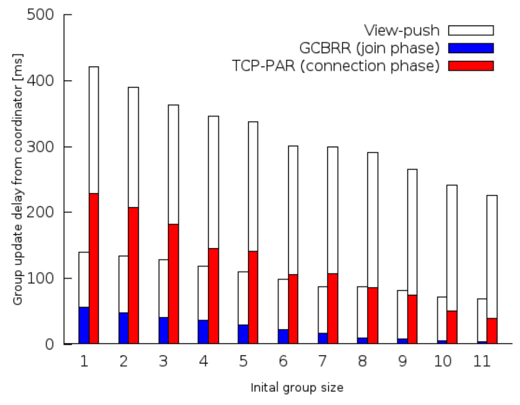


Fig. 3: Node join delay (top), and group update delay (bottom), when starting from initial group size N and adding $12 - N$ nodes in one shot.

leave the group, it closes the TCP connections to each group member. Ordinary processes detect that the connection was closed, elect the new coordinator and open a connection to it. The coordinator performs a view-push as in GCBRR, but in this case the group view is sent to each member over the respective TCP connection. As an implementation detail, we note that the coordinator uses a single thread for accepting new connections via a network event loop based on the *poll* Linux system call, similar to the technique described in [10].

To chart the cost of the join operation as a function of the group size and number joining processes, we conduct a series of experiments. In each experiment, the group starts with a different initial size, and then more nodes join so that the group ends up with a total of 12 nodes. The new nodes join the group following a single join poll. At each joining node, we record the time between the reception of the poll message until the reception of the group view from the coordinator. We also measure the time it takes for the coordinator to receive all join/connection requests and push the updated view to all members. The results are shown in Figure 3.

With both approaches the join delay naturally decreases as fewer nodes try to join the group. However, the average node join delay and group update delay for GCBRR is only $1/3$ that of the TCP-based approach. This is because the TCP variant has an extra connection setup overhead, which becomes significant when the number of joining nodes is large. Also, it sends the group view to each member separately, whereas GCBRR does this with a single transmission for old group members.

The leave operation for ordinary group members is very fast, and the overhead negligible with both approaches. However, in case of a coordinator departure, in the TCP variant every ordinary process must open a connection to the new coordinator. The corresponding overhead follows the same trend as in the join experiments discussed above (it is not shown here for brevity). In contrast, the cost for such a coordinator switch in GCBRR is practically zero. Of course, in both approaches, the new coordinator may have to perform a view push.

VII. Related Work

The reliability issues of multicast communication patterns over a shared medium have been studied mainly in the lower level of the network stack.

In [11] authors propose two reliable multicast protocols on top of IEEE 802.11 mechanisms. Both protocols assume a base station in the center of a microcell and one single sender at a time. The first protocol (LBP), is a leader based protocol while the other (PBP) is a probabilistic feedback-based protocol. In LBP, a sender claims the channel and sends a broadcast message to the receivers in the group where an elected leader responds with an acknowledgement. The rest of the receivers respond immediately with a negative acknowledgement in case of erroneous reception to destroy the acknowledgement of the leader causing a retransmission from the sender. In PBP a sender before the data transmission sends a multicast-RTS and waits to receive a CTS from each participant receiver. The receivers back-off with a certain probability based on the group size before the CTS transmission to avoid collisions. Both protocols increase broadcast reliability, however they do not guarantee collision avoidance and require strict clock synchronization.

A similar approach with focus on fairness is followed in LM-ARF [12] where an elected leader acknowledges mulitcast transmissions with the use of CTS/RTS frames. Moreover, LM-ARF performs an adaptation of the transmission rate based on the ARF [13] scheme of 802.11 to avoid inefficient broadcast transmissions when network saturates to its capacity. LM-ARF protocol inherits the time synchronization requirements from LBP and is closely coupled to the underlying MAC protocol.

The Broadcast Medium Window method (BMW) [14] provides a reliable broadcast solution where the nodes maintain lists for their neighbors and for the sequence numbers of the missing data packets. Each sender performs a collision avoidance phase and sends an RTS to inquire the missing messages from its neighborhood. In turn, each neighbor responds with a CTS that contains the requested information and the sender performs the transmission of the data. Each neighbor updates the local protocol state by overhearing the CTS/data exchange. BMW provides a reliable solution for broadcast, but requires a large number of messages and creates contention phases equal to the number of neighbors.

Complementary to the BMW method is the Batch Mode Multicast MAC (BMMM) [15] which makes use of a new frame type called AK (Request for ACK) in order to eliminate the multiple contention phases of BMW to a single one. The RAK frame follows directly after the transmission of the RTS, instructing the receivers to send their CTS and ACKs in order. BMMM follows an approach similar to our protocol with respect to the coordinated return of the replies, but involves more messages and abandons reliability when the network is stressed.

BSMA [16] approach improves the broadcast reliability assuming that the underlying radio hardware supports the DS (direct sequence) capture capability in order to lock-on a strong signal in the presence of interference. The protocol utilizes the 802.11 RTS/CTS frames to eliminate collision avoidance and makes use of negative acknowledgments for prompting re-transmissions.The results show that throughput and reliability increases over the traditional unreliable broadcast but the protocol relies strictly on 802.11 collision avoidance and requires specific radio capabilities. In the opposite, GCBRR is MAC-neutral while it is not based on any specific radio functionality.

The works above improve the physical broadcast reliability, however they do not have predicable performance which is a useful property for many latency sensitive applications. They also target general one-way multicast communication where the receiver merely sends a low-level acknowledgement, as opposed to GCBRR which works in an end-to-end fashion [17] and where replies may carry application-level information.

VIII. Conclusions

We propose a protocol for coordinated request-reply and group management in wireless systems, which eliminates contention and exploits the broadcast nature of the shared medium to minimize the number of message transmissions and latency. The proposed protocol does not rely on any advanced MAC features, and can work on top of different networking technologies.

Our evaluation over 802.11 WiFi for scenarios where the system operates close to the network capacity, shows that the proposed protocol achieves significantly higher throughput, lower latency and better predictability than unicast-based point-to-point approaches. These results are encouraging given that WiFi comes with a very effective flow-control mechanism and MAC-level support for unicasts. Note that, in case of message loss (e.g., due to external interference) the proposed protocol would perform even better compared to unicast-based approaches, as it requires fewer packet transmissions.

In the future we will experiment with simpler radios, and plan to investigate the usage of this protocol to support symmetrical process groups with N-to-N message flows.

References

[1] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 731–736. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837461

[2] P. Sujit, D. Kingston, and R. Beard, "Cooperative forest fire monitoring using multiple UAVs," in *2007 46th IEEE Conference on Decision and Control*. Institute of Electrical & Electronics Engineers (IEEE), 2007.

[3] C.-S. Lee, K.-H. Lee, and J.-K. Lee, "A group RPC protocol for distributed systems," in *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat. No.97TH8237)*. Institute of Electrical & Electronics Engineers (IEEE), 1997. [Online]. Available: http://dx.doi.org/10.1109/ICICS.1997.652090

[4] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi, "Team-level programming of drone sensor networks," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '14. New York, NY, USA: ACM, 2014, pp. 177–190. [Online]. Available: http://doi.acm.org/10.1145/2668332.2668353

[5] M.-C. Chan, C. Chen, J.-X. Huang, T. Kuo, L.-H. Yen, and C.-C. Tseng, "OpenNet: A simulator for software-defined wireless local area network," in *2014 IEEE Wireless Communications and Networking Conference (WCNC)*. Institute of Electrical & Electronics Engineers (IEEE), apr 2014. [Online]. Available: http://dx.doi.org/10.1109/WCNC.2014.6953088

[6] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[7] http://nitlab.inf.uth.gr/NITlab/index.php/nitos.html, [Online; accessed 28-Jan-2016].

[8] http://nitlab.inf.uth.gr/NITlab/index.php/hardware/wireless-nodes/icarus-nodes, [Online; accessed 28-Jan-2016].

[9] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in the air: Practical wireless network coding," *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 497–510, June 2008. [Online]. Available: http://dx.doi.org/10.1109/TNET.2008.923722

[10] A. Chandra and D. Mosberger, "Scalability of linux event-dispatch mechanisms," in *In Proceedings of the 2001 USENIX Annual Technical Conference*, 2001, pp. 231–244.

[11] J. Kuri and S. Kasera, "Reliable multicast in multi-access wireless LANs," in *IEEE INFOCOM 99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies.* Institute of Electrical & Electronics Engineers (IEEE), 1999. [Online]. Available: http://dx.doi.org/10.1109/INFCOM.1999.751463

[12] S. Choi, N. Choi, Y. Seok, T. Kwon, and Y. Choi, "Leader-based rate adaptive multicasting for wireless LANs," in *IEEE GLOBECOM 2007-2007 IEEE Global Telecommunications Conference*. Institute of Electrical & Electronics Engineers (IEEE), nov 2007. [Online]. Available: http://dx.doi.org/10.1109/GLOCOM.2007.694

[13] A. Kamerman and L. Monteban, "WaveLAN®-II: a high-performance wireless LAN for the unlicensed band," *Bell Labs Tech. J.*, vol. 2, no. 3, pp. 118–133, sep 1997. [Online]. Available: http://dx.doi.org/10.1002/bltj.2069

[14] K. Tang and M. Gerla, "MAC reliable broadcast in ad hoc networks," in *2001 MILCOM Proceedings Communications for Network-Centric Operations: Creating the Information Force (Cat. No.01CH37277)*. Institute of Electrical & Electronics Engineers (IEEE), 2001. [Online]. Available: http://dx.doi.org/10.1109/MILCOM.2001.985991

[15] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai, "Reliable MAC layer multicast in IEEE 802.11 wireless networks," in *Proceedings International Conference on Parallel Processing*. Institute of Electrical & Electronics Engineers (IEEE), 2002. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2002.1040910

[16] K. Tang and M. Gerla, "Random access MAC for efficient broadcast support in ad hoc networks," in *2000 IEEE Wireless Communications and Networking Conference. Conference Record (Cat. No.00TH8540)*. Institute of Electrical & Electronics Engineers (IEEE), 2000. [Online]. Available: http://dx.doi.org/10.1109/WCNC.2000.904675

[17] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402