ROCC: A Communication Overlay Abstraction for Coordination Middleware

Seth Holloway
Christine Julien

TR-UTEDGE-2007-002

ROCC: A Communication Overlay Abstraction for Coordination Middleware

Seth Holloway and Christine Julien

Mobile and Pervasive Computing Group
The Center for Excellence in Distributed Global Environments
The University of Texas at Austin
{sethh, c.julien}@mail.utexas.edu,
http://mpc.ece.utexas.edu

Abstract. As wireless mobile devices become more popular, the potential for new collaborative applications has emerged. For example, a group of coworkers can opportunistically come together, and share project documents and other data. Such applications are characterized by the fact that information needs to be distributed among all members of a group, while providing robustness in the face of network disconnections and reconnections. Supporting such coordination currently requires using low-level multicast protocols that are complex and resource intensive to initialize and maintain without the support of an infrastructure. In this paper, we introduce the ring overlay for collaborative coordination (ROCC), which offers a fair, reliable coordination service for dynamic environments. ROCC is tailored to networked environments characterized by frequent topology changes, and the abstraction allows applications to seamlessly adapt and operate in the face of these changes. By providing application-level communication, ROCC specifically addresses application situations requiring group coordination. ROCC is influenced by traditional token ring and overlay communication protocols but leverages unique properties of the broadcast nature of the wireless environment to improve performance. The result is a minimal, fair, dynamic group coordination service. This paper introduces the ROCC abstraction, an implementation, and an analysis.

1 Introduction

The combination of heightened numbers of mobile electronic devices and users' acceptance of pervasive computing hardware and applications has enabled new classes of mobile, collaborative applications. These new applications demand increased levels of coordination as we strive to maintain connectivity in an always-on world. Mobile collaborative applications can be envisioned in lecture halls, where students and teachers share notes and lecture information, to coffee shops, where an ad hoc group of customers can initiate a multi-player game that requires dynamically sharing changing state information.

These collaborative applications share a number of characteristics that engender them to the creation of a generic coordination infrastructure. We have

created such an infrastructure, Sliverware [6], that encapsulates software and communication concerns associated with collaborative applications in dynamic mobile computing environments. In addition to requiring low-level communication capabilities (typically provided by the wireless medium in mobile scenarios), collaborative applications require a group coordination component that determines how and when information is shared among participants. While existing protocol approaches can be manipulated to support mobile collaborative applications, either their performance suffers because they are not tailored for this domain or applications have to perform extra communication tasks because the the protocols are not particularly suited to the application's characteristics.

In this paper, we create a novel group coordination abstraction targeted to supporting the kinds of information sharing necessary in collaborative applications. This abstraction and middleware service, the ring overlay for collaborative coordination (ROCC), accounts for the dynamic departure and addition of participants in a collaborative activity. In addition, ROCC takes advantage of properties both of the application (e.g., the definition of the communicating partners) and of the physical communication channel (e.g., the wireless medium) to provide guaranteed delivery, fairness, and reduced overhead. ROCC is suited to supporting applications in which all information shared within the collaboration group must be received by all members of the group, and the ability to transmit data to group members must be allocated fairly. To achieve these goals, the ROCC abstraction leverages aspects of token ring protocols, application-level overlay protocols, and existing coordination languages.

The remainder of this paper is organized as follows. In Section 2, we provide a combination of a high-level, informal description of the ROCC protocol and its underlying formalization. In Section 3, we briefly describe our prototype implementation and present an analysis of its performance. Section 4 places the ROCC protocol in the context of related work, and Section 5 concludes.

2 Coordinating Collaboration Participants: The ROCC Abstraction

Collaborative applications must create and coordinate groups of participants. Groups must remain connected so that users can reliably exchange information necessary to support collaborative activities. Within any single collaborative application, multiple groups may exist and must be maintained independently. In a classroom, one group may include all students and the teacher; other groups may support teams working together on class projects.

As mobile computing becomes more widely accepted, mobile ad hoc networks will be increasingly used to support collaborative applications. Mobile ad hoc networks form opportunistically in response to devices' movements in physical space. When devices are close enough to create a wireless link, a new connection is added. In such environments, communication protocols provide minimal point to point and multicast abstractions. The latter can support coordination among collaborative groups when all members of the group have joined the multicast

group, but the implementations in mobile ad hoc networks incur significant overhead, as they are based on protocols that functioned well in networks with significant centralized infrastructure but are not well suited to dynamic networks. In addition, when many participants try to send information simultaneously, such communication strategies can result in significant contention for the wireless medium, requiring additional negotiation protocols for group communication.

The principle motivation of this work is to create a communication abstraction that directly reflects the requirements of collaborative applications in dynamic environments. Our approach raises the level of abstraction for the application developer by directly reflecting the structure and communication pattern of the group. The fundamental structure of our abstraction is a ring overlay, reminiscent of token ring networks. Each node in the collaboration group can exchange information only when it possesses the token. Imposing such an overlay gives each group member a chance to transmit and, as described below, can aid in ensuring that every member receives every other member's transmissions.

Fig. 1 depicts such an overlay and shows that it may not necessarily directly reflect available physical connections. Instead, the overlay's implementation may use multiple hops in the physical network to provide what appears to be a single hop connection in the overlay. If the ring does reflect a subset of the network's true topology, we avoid unnecessary transmissions, which can serve to increase the overall efficiency and speed. Overall, providing the application with the appearance of this ring structure directly matches communication capabilities to collaborative applications' expectations of underlying group communication schemes and prevents collaborative application developers from having to awkwardly manipulate existing protocols to achieve their goals.

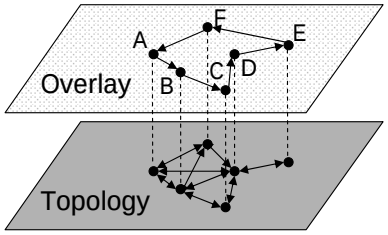

Fig. 1. A sample network demonstrating connectivity between nodes.

In the remainder of this section, we describe the ROCC model. We first discuss how the ring is used to support group coordination. Then we provide details of the ring maintenance aspects, including adding, removing, and maintaining group members. In our subsequent discussion of ROCC, we assume the presence of only one application group. If multiple application groups exist, we assume the ROCC overlay runs separately for each one, on a different (non-interfering) channel to prevent collisions between the groups. For now, we assume the assignment of these channels to groups is performed out of band; future work will investigate how to handle dynamic channel assignments.

2.1 Basic Operation in the Ring Overlay

Before looking at how ROCC constructs and maintains its overlay, we first look at how collaborative applications use ROCC to ensure fair and reliable collabo-

ration. ROCC establishes and maintains an overlay on top of the true topology; in ROCC, this overlay is presented to the application as a directed token ring network. Once the node is part of a ring, the node can communicate only when it possesses that ring's token. By imposing ring-like behavior, ROCC significantly reduces the likelihood of competition for the physical medium and the potential for interference between transmissions from members of the same group.

In ROCC's overlay ring, each node has a predecessor (the previous node in the ring) and a successor (the next node in the ring). When a ring consists of only a single node, the node is its own predecessor and successor. Fig. 2 shows some simple examples. It is important that nodes adhere to this structure even in the simplest cases to ensure that adding and removing nodes can be done smoothly.

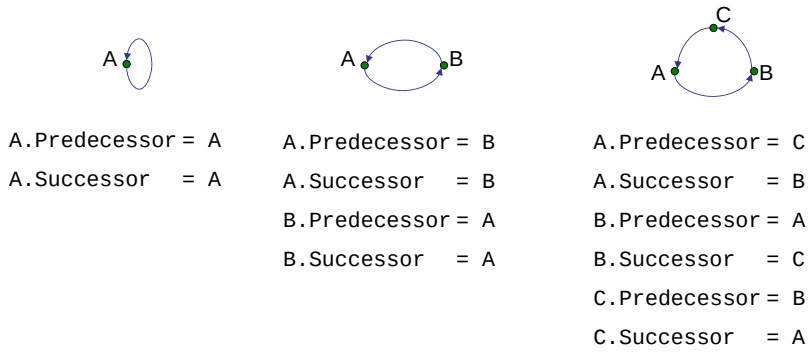

Fig. 2. Successors and Predecessors for three basic networks.

The goal of data transmission in ROCC is to ensure that, after each complete token rotation, every node in the ring has the exact same picture of the data as every other node. When forwarding the token, each node forwards a list of all other nodes in the group. Token carrying messages are structured in the following format:

$$\langle ring\_address,\ source,\ destination,\ participants,\ data_i \rangle$$

The *ring_address* identifies the ring, while the *source* and *destination* addresses indicate the sender and targeted receiver of this packet. The *participants* list contains the identities of the members of the ring. This portion of a ROCC message is therefore variable in length, depending on the number of ring participants and has the following format:

$$participants \equiv \langle node\_address_1,\ \ldots\ , node\_address_n \rangle$$

The final portion of the message, $data_i$ contains this round's data for the sending node (i.e., node i), if the node has data to send.

To optimize its performance, ROCC takes advantage of the inherent broadcast nature of wireless links. This means when a node receives the token, it may

have already overheard data transmissions between other nodes. Consider Fig. 1. If node A transmits data within its token message to its successor (B), D and F will overhear the message because these nodes are within communication range of A. Since C is not within range of A, B will transmit A's data to C. C will not forward A's data to D since D overheard the original transmission.

ROCC nodes take advantage of this optimization by storing any data contained in overheard packets. When a node eventually receives the token, it compares the token's list of ring participants to the already buffered (overheard) data. The node holding the token proactively solicits any missing data from the predecessor. The result of this solicit message is an array of the missing data, and other nodes may overhear this exchange and store this information, reducing the future overhead. Especially in densely connected networks, this drastically reduces the communication overhead and, consequently, the speed at which the token can circulate. This is particularly practical in collaborative applications that demand that every node receive every other nodes' data transmissions. Implementing such a situation using traditional communication protocols at the application level results in numerous retransmissions due to a significant potential for collision, as evaluated in the next section.

If a packet that a node overhears contains a ring address other than the node's ring address, the node is marked in the connectivity table, and it raises an internal flag to merge the ring with this newly discovered ring. After receiving the token but before updating the neighbor information as described above, the node begins the ring join phase, which will be described in the next subsection.

The ROCC protocol as described above does not completely prevent all collisions, as simultaneous transmissions in neighboring rings may compete with each other. In overhearing these transmissions, ROCC nodes can discover new nearby rings; the detriment is the expense of the overhead of recovering from collisions. However, the degree of collisions is likely significantly decreased in comparison to straightforward contention for the media; this aspect will be analytically evaluated in Section 3.

Formal Description of Ring Operation. To formally describe nodes' behavior in ROCC, we use I/O Automata notation [7]. We show the behaviors of node A, indicated by the subscript A. Each *action* has an effect guarded by a precondition. Actions without preconditions are *input actions* triggered by another host. Every action executes as one atomic step. We abuse I/O Automata notation slightly by using "send *TokenPacket* to B" to indicate a sequence of actions that triggers the action TokenPacketReceived on node B. Fig. 3 provides a listing of the state information stored by each node. The formal description also references a set of helper methods; these are detailed in Fig. 4.

Fig. 5 shows the action TokenPacketReceived for node A. Within the loop that specifies the proactive solicitation, we model the request for each node's data as a separate SolicitData message; in the implementation, these are encapsulated into a single request, as are the replies from the predecessor. The

ringAddress	the address of the ring the node is participating in
predecessor	the previous node in the ring
successor	the next node in the ring
participantsTable	the set of nodes known to be participating in the ring
dataTable	buffered data for the participants that has been overheard from other nodes' transmissions
oldBuffer	the last round's data; saved in case the successor solicits data
waitingForAck	boolean indicating that this node has transmitted the token but is waiting for an acknowledgement
ackTimer	timer for token acknowledgement; if no acknowledgement is received before the time is expired, the node assumes the neighbor is lost
join	flag that indicates whether this node has discovered a nearby neighbor from a different ring
discoveredNeighbor	address of one-hop neighbor that is not a member of this ring
attemptJoin	condition that, when true, enables this node to attempt to join with a nearby ring
joinRequested	condition set when another node requests a join
joinRequester	the address of the node requesting the join
acceptJoin	set to true by this node when it determines a willingness to participate in the requested join
departing	flag indicating this node's intention to announce its departure

Fig. 3. ROCC State Information

updateParticipants(H[])	replaces the current list of participants with those referenced in the passed array
updateBuffer(i, d)	for the ring participant identified by i, add the data d to the buffer
bufferComplete()	return true if the buffer contains data for all of the participants
getSuccessor(i)	retrieve the successor of the node i from *participants*

Fig. 4. ROCC Helper Methods

ackTimer is used to handle failures; its behavior and the actions associated with joining and departing will be described in subsequent sections.

Fig. 6 shows the action a node takes when it receives an acknowledgement that its successor received the token. This action simply cancels the *ackTimer* and resets the local *waitingForAck* variable.

Fig. 7 shows the action that occurs when a node receives a solicit data message from its successor. As above, we model these as separate requests, but for a given node, all of the data will be encapsulated in a single message to save overhead. This is the same message that is used to handle overhearing and buffering data sent between other source/destination pairs.

```
TokenPacketReceived_A(p)
  Effect:
    if p.ring_address = ringAddress then
      if p.destination = A then
        predecessor = p.source
        send AcknowledgeToken to predecessor
        updateParticipants(p.participants)
        if departing then {see Section 2.3}
          send SetSuccessor(successor) to predecessor
          send SetPredecessor(predecessor) to successor
          [empty all state associated with ring]
        else if joinRequested then {see Section 2.2}
          acceptJoin = true
        else if join then {see Section 2.2}
          attemptJoin = true
        else
          for each i ∈ participantsTable do
            if dataTable(i)=∅ then
              send SolicitData(i) to predecessor
          if bufferComplete() then
            [deliver data to application]
            oldBuffer = dataTable
            dataTable = ∅
            send TokenPacket to successor
            waitingForAck = true
            ackTimer.start
      else
        updateBuffer(p.source, p.data)
```

Fig. 5. Receiving a Token Packet

```
AcknowledgeTokenPacketReceived_A(p)
  Effect:
    successor = p.source
    waitingForAck = false;
    ackTimer.cancel;
```

Fig. 6. Acknowledging a Token Packet

Once a node has solicited data from its predecessor, it must wait for all of that data to be received before forwarding the token. Fig. 8 shows the action that is executed when a solicited (or overheard) data packet is received.

2.2 Building Ring Overlays

Because, in the base case, a node by itself is also a ring, adding a new node to an existing ring and joining rings together uses the same process. When a node is

```
SolicitDataReceived_A(p)
  Effect:
    if p.destination = A then
      send DataPacket(oldBuffer(p.i)) to successor
```

Fig. 7. Receiving a Data Solicitation

initialized into a ring with a single member, it uses its unique id (e.g., IP address or MAC address) as the ring's identifier.

To discover a new ring, a node within a ring must receive or overhear a transmission from a nearby ring. Because rings containing only a single node do not actually require data transmission, such self rings periodically signal their presence with a beacon. Nodes in rings with more than one node do not have to do this because their standard transmissions can be overheard by nearby nodes. When a nearby node hears either type of transmission (a beacon from a single-node ring or a standard packet from a ring other than its own), it sets a flag to join the neighboring group.

```
DataPacketReceived_A(p)
  Effect:
    updateBuffer(p.source, p.data)
    if p.destination = A then
      if bufferComplete() then
        [deliver data to application]
        oldBuffer = dataTable
        dataTable = ∅
        send TokenPacket to successor
        waitingForAck = true
        ackTimer.start
```

Fig. 8. Receiving a Data Packet

Within the standard ring operation described above, each node will eventually receive and acknowledge reception of the token, at which point it checks the join flag. If the join flag is set, the node sends a join request to the neighboring ring. This join request contains the sending node's address, the sending node's ring address, the sending node's predecessor and successor, and the destination ring's identifier. Since only a ring's token holder is allowed to transmit information, the receiving node in the newly discovered ring must also hold its ring's token for a join operation to commence. For this reason, the join request is followed by a brief delay to give the node in the neighboring ring some time to capture its ring's token. If the neighboring node does not reply within the allotted time, the requester returns to normal operation by transmitting data and forwarding the token.

This join process does not ensure that nearby rings join in a timely fashion, but it does avoid the deadlock that arises when a node in one ring has blocked its ring to start a join, while a different node in the target ring has also blocked its ring to start a join. This could also be avoided using extra control messages, but

these messages cause more possibilities for collisions and increased coordination overhead.

If the discovered node in the neighboring ring captures the token in time, it will send an acknowledgment with its predecessor and successor. At this point, the ring that has the lower ring identifier becomes the control point for the join operation. The control node swaps the successors from the joining nodes (itself and the active node from the joining ring) by sending an update messages to the affected nodes (both the new successors and new predecessors). Fig. 9 demonstrates the three cases that can occur during the join operation. In all of these cases, assume nodes B and C moved within communication range of each other. Each case exchanges B and C's successors. The third case is the general case for joining two rings, since there can be arbitrarily many nodes between A and B and between D and C.

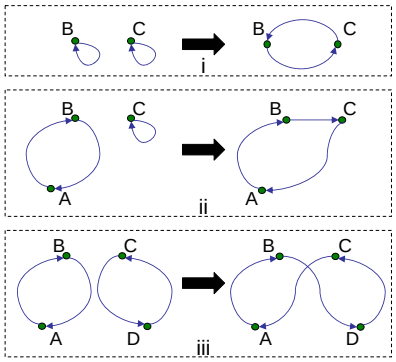

Fig. 9. The three possible configurations for B joining C: i) single node to single node, ii) single node to group, iii) group to group.

After completing a join, the node that had to change its ring address (*not* the control node) sends an update message to its original ring.

Formal Description of Ring Creation The first step of our formal description checks the *join* flag in the TokenPacketReceived method as shown in Fig. 5. This enables the AttemptJoin action, shown in Fig. 10. The attempt to join with a neighboring ring creates a JoinRequestPacket that it sends to the *discovered neighbor*.

The *joinTimer* waits for a response from the node in the neighboring ring. If the timer expires, it is assumed that the neighboring node either departed or could not retrieve its ring's token in time, and the state regarding this join is canceled. If the node rediscovers this neighboring ring during a subsequent round, the join request will be reattempted. The CancelJoinAttempt action, shown in Fig. 11, is enabled when the *joinTimer*

AttemptJoin$_A$
  Precondition:
    join $\wedge$ *attemptJoin*
  Effect:
    send JoinRequestPacket to *discoveredNeighbor*
    joinTimer.start

Fig. 10. Attempting to Join a Neighboring Ring

expires and resets the state associated with the join attempt. The node then continues in the same manner as the TokenPacketReceived action in Fig. 5 would have had the join not been attempted.

When the node in the neighboring ring receives a JoinRequestPacket, it sets the *joinRequested* flag. The TokenPacketReceived action, before attempting to initiate a join, checks this flag. If it is set, the TokenPacketReceived action sets the *acceptJoin* flag, which enables the AcceptJoinRequest action, as shown in Fig. 12.

```
CancelJoinAttempt_A
  Precondition:
    joinTimer.expired
  Effect:
    join = false
    discoveredNeighbor = ∅
    [solicit data and forward token as in Fig. 5]
```

Fig. 11. Cancelling a Join Attempt

```
AcceptJoinRequest_A
  Precondition:
    joinRequested ∧ acceptJoin
  Effect:
    send AcceptJoinPacket to joinRequester
    if ringAddress < joinRequester.ringAddress then
      send setPredecessorPacket to successor
      send SetSuccessorPacket to joinRequester
      successor = joinRequester.successor
      [solicit data and forward token as in Fig. 5]
    [reset local flags dealing with joins]
```

Fig. 12. Accepting a Join Request

The counterpart to the above action, AcceptJoinPacketReceived has similar functionality, and is omitted for brevity. In both cases, the node that is part of the ring with the lower ring address initiates the swap of predecessors and successors, and this node also continues with the token. The other node's setSuccessorPacketReceived action is initiated, which generates a setPredecessorPacket for the node's old successor and resets the node's successor. This node also propagates a ResetRingAddressPacket around its old ring. The handling of these last few actions is simple and straightforward, and they are also omitted for brevity.

2.3 Maintaining Ring Overlays in the Face of Failures

When coordination becomes no longer necessary, group members will start to depart, and the ring will begin to dissolve. In addition, as nodes move in the

wireless environment, nodes may incidentally become disconnected from the ring. In general, nodes intending to disconnect from the group will announce this intention before simply disconnecting. When this occurs, the node sets a local flag indicating this intention. When the node receives the token, it handles its own departure by setting up a connection between its predecessor and its successor. This quickly and easily remotes the departing node and maintains the general group ordering.

Such an announced departure is the best case, but since ROCC is designed for supporting dynamic mobile communication, it must also handle unpredictable cases that result from device mobility and the unreliability of the wireless links. The remainder of this section details how ROCC handles these cases and makes their complexity transparent to the application.

Unannounced Disconnection. When a node departs from the ring unexpectedly, the network must restore the ring connectivity and gracefully discard information related to the departed node's participation. In ROCC, a departed node's predecessor will eventually attempt to forward the token to its successor (the departed node). When the successor does not acknowledge receipt of the token, the sender assumes the node has departed.

Because the sending node has complete information about the group members and their order (given the packet information above), the sender can simply select its successor's successor and attempt to send the packet directly there. If this node acknowledges receipt of the token, it sets its predecessor to the sender, and the sender sets the node as its new successor. Because ROCC relies on an underlying routing protocol, it does not matter if this new successor is directly connected to the sending node or not; the underlying communication substrate will handle the end-to-end connections. If the successor's successor is unavailable (i.e., it's also been disconnected) the node holding the token simply continues down the list of the ring's participants (in order) until it finds a node that is responsive.

```
AckTimerExpired_A
  Precondition:
    ackTimer.expired
  Effect:
    successor = getSuccessor(successor)
    if successor != A then
      send TokenPacket to successor
      waitingForAck = true
      ackTimer.start
```

Fig. 13. Handling an Acknowledgement Timeout

Formally, when the *ackTimer* expires, the actionAckTimerExpired, shown in Fig. 13 is enabled. If the *ackTimer* expires again, the process continues, until the next available successor is this node.

Lost Ring Address. ROCC is almost completely decentralized. However, to ensure that ring addresses are unique, the address assigned to the ring needs to be the address of one of the nodes in the ring. When this node departs

unannounced, this ring address needs to be updated. In the process described above for handling unannounced disconnections, if a node discovers that it has lost its successor *and* that successor's address is also the ring address, the node sends a ResetRingAddressPacket around the ring before passing the token along to the next neighbor. This node changes the ring's identifier to its own addresses. We omit the formalization of handling a lost ring address for brevity.

Lost Token. The last and most difficult case to handle occurs when a node departs without announcement while it is holding the token. In this case, the rest of the ring is completely crippled. In handling this case, we must balance the overhead of reorganizing the group with the increased latency incurred by the increase in the amount of time the group does not communicate at all. Since this token loss is a rare event, it is appropriate for ROCC to ensure that the token is lost before generating a new one. If a node does not receive the token in twice its estimated token rotation period (based on the average time it took the token to rotate for the past rotations), it queries its predecessor to see if its still alive. The node continues to query successive predecessors until it either finds one alive or finds itself. If this process does not locate the token, it is assumed that the token disappeared with one of the departing nodes, and the remaining nodes coordinate to generate a single new token.

3 An Analytical Comparison of ROCC and Multicast

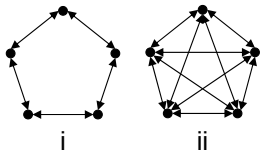

Fig. 14. The two types of network topologies examined analytically

To gauge ROCC's usefulness as a coordination middleware service, we analytically compared it to another coordination service, wireless multicast. Variations on wireless multicast (such as MAODV [10]) are the most commonly used communication service for support of coordination middleware, especially when considering collaborative applications. In our analysis, we compared the latency and overhead of the two approaches. To bound our analysis, we compare the approaches' performance for two types of networks (as shown in Fig. 14): a network in which each node is connected only to two other nodes, and a fully connected network in which every node is directly connected to every other node. The picture shows five nodes in each network; our analysis varies the number of nodes in the network from 2 to 20. These represent two fairly extreme cases; we expect that results for other types of topologies lie in between. We expect that ROCC can take full use of overhearing optimizations when there are several overlapping connections, so we expect the second type of setup to fully demonstrate ROCC's potential. Both types of configurations shown in Fig. 14 are possible in real networks, but in en-

vironments where collaborative applications are likely, we expect networks with several redundant links to be common.

Both ROCC and basic multicast run with the support of media access control (MAC) protocols. We assume that both ROCC and multicast use the 802.11 MAC protocol without the RTS/CTS exchanges that are commonly used to reduce collisions. We omit RTS/CTS exchanges because they have been shown not to provide a benefit to throughput in wireless ad hoc networks (and may in fact be detrimental) [11] due to the fact that a node's interference range is much larger than the transmission range over which the RTS/CTS functions. We assume a bandwidth of 2Mbps, and we evaluate the latency and overhead for each approach for an entire "round" of collaboration. We assume an active collaboration activity in which every participant has (a 1KB piece of) data to send, and we determine the amount of time and overhead involved in ensuring that every participant has every other participant's piece of data. For now, we evaluate only the costs of participating in the ring; future work will extend this analysis to include situations in which nodes must be added and removed from the group during normal function. In ROCC, this will happen as described in the

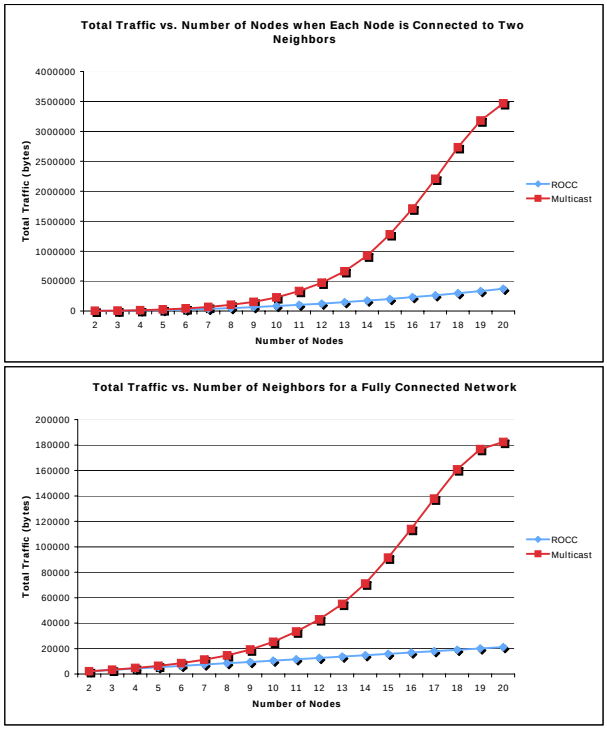

Fig. 15. The total amount of traffic generated by ROCC and multicast for each network configuration.

previous sections; multicast also incurs overhead in setting up and maintaining routes in the face of dynamics.

Fig. 15 shows the total data sent in the two topologies when transmitting every node's data to every other node. ROCC generates far fewer bytes than a multicast protocol; this can be explained by two factors: ROCC eavesdrops to help reduce the amount of coordination necessary and ROCC transmissions do not experience collisions. Overhead for ROCC is modeled as described in Section 2 while multicast operation is modeled based on protocols similar to MAODV [10]. As described in the previous section, upon receiving the token a node using ROCC will request information that it has not already heard. In the case of full connectivity, a node does not have to request anything because it has already overheard all of the other participant's transmissions. When the network is less than fully connected, ROCC follows the same procedure; however, each node will request the missing information, then send their own data and the token to their neighbor. Each node in ROCC transmits one request for missing information, one individual data packet, and one message containing the data that the successor requested. Multicast protocols must send one message to every member of the group and send along the multicast tree. As a result, mul-

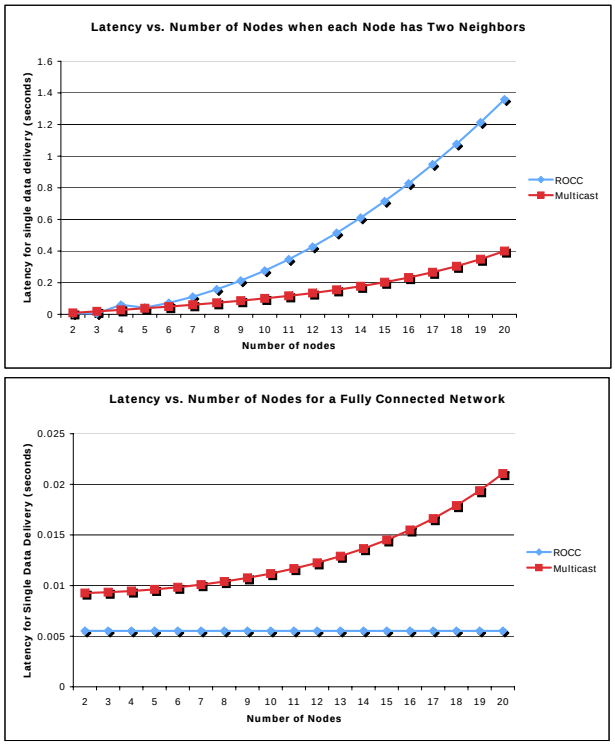

Fig. 16. The latency for one node's data to reach the entire network.

ticast protocols generate more messages as the number of connections decreases, though the messages ROCC sends are likely to be larger (since they contain an aggregation of several nodes' data items). Multicast encounters collisions when neighboring nodes attempt to transmit at the same time. ROCC forces nodes to take turns, eliminating the possibility of collisions. We model only the interference caused by direct neighbors in the multicast case; in truth, neighbors multiple hops away can also interfere [11].

Fig. 16 shows the delay between when a node sends its data until every other node in the network receives that data. To send a single node's data to the entire group in a sparsely connected network (e.g., one shaped as a ring), ROCC must pass the token to all members in order. Multicast transmits data directly to all members—the only slow down incurred is due to collisions. This comparison is depicted in the upper graph in Fig. 16. However, when a ROCC node wants to send its data to all members of a fully connected network, ROCC provides ordered communication and allows the node to send the data only once since everyone else will overhear it. To send one node's data to everyone in a fully connected network using multicast will generate collisions proportional to the number of nodes within communication range. As the number of nodes increases, the amount of collisions forces many retransmissions that greatly slow the network. This demonstrates the benefit ROCC achieves due to overhearing. In collaborative application domains, it is likely that group participants are nearby (e.g., in the same classroom), so the likelihood for redundant connections is high. ROCC takes advantage of these redundant connections, while traditional approaches such as multicast are crippled by them.

Overall, our analysis shows that ROCC provides low latency communication in additional to decreased overhead; these properties make ROCC an ideal protocol for coordination on wireless devices.

4 Related Work

ROCC provides high-level coordination constructs built on underlying communication protocols that collaborative applications can use for timely, reliable group communication. ROCC is inspired by token ring protocols whose goals revolved around mediating access to a shared medium. The Wireless Token Ring Protocol (WTRP) [4] brought token rings into the wireless domain. WTRP builds on 802.11, and it aimed to mediate access to the wireless medium. WTRP has been shown to outperform 802.11 by itself [5] while at the same time minimizing the delays applications experience. Extensions to WTRP [2, 3] focus on minimizing energy consumption or maximize reliability. While these protocols provide inspiriation for ROCC, the fundamental goal is different. ROCC relies on underlying protocols for medium access control and instead aims to provide coordination constructs tailored for collaborative applications. As such, ROCC aims to ensure reliable delivery to every group members while existing token ring approaches are tailored to supporting unicast communication.

As a potential middleware service for coordination, ROCC has the potential to replace existing approaches within coordination middleware. For example,

the LIME middleware [9] uses basic multicast communication to ensure delivery among the members of a LIME coordination group. As demonstrated by the analysis in the previous section, replacing multicast in LIME has the potential to offer decreased overhead and increased timeliness for group-wide communication. In the TOTA middleware [8] propagation rules require a distributed overlay data structure to ensure that tuples are received at every member of a group of TOTA nodes. Employing ROCC in this situation may be able to produce the same benefits as underneath LIME, providing an abstraction on which this distributed data structure can be implemented. Decentralized publish-subscribe systems (e.g., [1]) connects groups of publishers and subscribers and aims to ensure that a publisher's event is delivered to every registered subscriber. When the rate of publication is high, ROCC can offer significant benefits to such applications by ensuring this delivery with minimal overhead. Finally, middleware for collaborative applications, e.g., [6], often entail group definitions in which groups of collaborators should maintain a consistent overall picture. Using ROCC as the group support protocol in such middleware provides a clean abstraction for this grouping mechanism, while also offering the benefits of reliability, timeliness, and decreased overhead.

5 Conclusions

Thanks to the proliferation of wireless devices, there are more ways to share information with one another than ever before. Exciting new applications push the envelope and extend the role of coordination. To aid in this exciting time we have presented a middleware service, the ring overlay for collaborative coordination (ROCC), which we designed to support the kind of information sharing found in collaborative applications executing in modern mobile environments.

ROCC combines properties of applications and the communication channel to create a decentralized, efficient coordination service for dynamic environments. Since ROCC builds on token rings, we benefit from the fairness inherent in turn-taking schemes. The turn-taking also frees communication from collisions. ROCC coordinates the underlying devices providing a solid communication foundation. An analysis of ROCC shows that the protocol outperforms existing coordination services with significantly lower overhead and decreased latency. These results demonstrate that, as applications' demands on coordination continue to grow, we must adopt efficient coordination abstractions that can both ease the development task by raising the level of abstraction and can inherently lead to efficient implementations.

Given the feasibility of the ROCC approach demonstrated in this paper, future work will provide further analysis of dynamics associated with ROCC, comparisons through simulation, and the encapsulation of ROCC as a standalone coordination middleware service that can be incorporated into both middleware and application solutions.

Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This work was funded, in part, by the National Science Foundation (NSF), Grant # CNS-0620245. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. A. Carzaniga, D. Rosenblum, and A. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the 19th Annual ACM Symp. on Principles of Distributed Computing*, pages 219–277, 2000.
2. Z. Deng, Y. Lu, C. Wang, and W. Wang. E^{2}WTRP: an energy-efficient wireless token ring protocol. In *Proc. of the 15th IEEE Int'l. Symp. on Personal, Indoor and Mobile Radio Communications*, pages 574–578, September 2004.
3. Z. Deng, Y. Lu, C. Wang, and W. Wang. EWTRP: enhanced wireless token ring protocol for small-scale wireless ad hoc networks. In *Proc. of the 2004 Int'l. Conf. on Communications, Circuits, andSystems*, pages 398–401, June 2004.
4. M. Ergen, D. Lee, R. Sengupta, and P. Varaiya. WTRP: Wireless token ring protocol. *IEEE Trans. on Vehicular Technology*, 53(6):1863–1881, November 2004.
5. M. Ergen, D. L. R. Sengupta, and P. Varaiya. Wireless token ring protocol-performance comparison with IEEE 802.11. In *Proc. of the 8th Int'l. Symp. on Computers and Communication*, pages 710–715, June/July 2003.
6. S. Holloway and C. Julien. Developing collaborative applications using sliverware. In *Proc. of the 14th Int'l. Conf. on Cooperative Information Systems*, volume 4275 of *Lecture Notes in Computer Science*, pages 587–604, 2006.
7. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
8. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Proc. of the 2nd Int'l. Conf. on Pervasive Computing and Communications*, pages 263–273, March 2004.
9. A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21st Int'l. Conf. on Distributed Computing Systems*, pages 524–533, April 2001.
10. E. Royer and C. Perkins. Multicast ad hoc on-demand distance vector (MAODV) routing. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1999.
11. C. M. Wu and T. C. Hou. The impact of RTS/CTS on performance of wireless multihop ad hoc networks using ieee 802.11 protocol. In *2005 IEEE Int'l. Conf. on Systems, Man and Cybernetics*, volume 4, pages 3558–3562, October 2005.