# Consistent Shared Memory Programming on a Collection of Mobile Devices

## Abstract

*Location-based services accessed by mobile devices are increasingly pervasive. Much of the data processed by such services are distributed near the locations where the processed results are needed. If the mobile devices could compute on this data directly, rather than accessing servers in the cloud for all non-local computation, they could gain three major advantages: (1) ease the bandwidth pressure on already overloaded access networks, (2) lead to quicker response times, and (3) improve battery lifetime of mobile devices. To realize the above vision, there needs to be a way to easily program a collection of mobile devices as a whole, since substantial programming complexities arise due to the mobile nature of such platforms.*

*In this paper, we present a programming abstraction layer that allows mobile devices to consistently access shared data. We argue that consistent shared memory (CSM) is a key requirement in enabling an easy-to-program, distributed mobile platform. Substantial challenges exist in realizing this vision, and the key to our approach is porting ideas of consistency and coherence from the parallel computing domain to the wireless domain. We illustrate, using a distributed car parking application, how applications that require data to remain reliable and be consistently accessible can now be enabled over mobile nodes. Our ns-2 evaluations of the car parking application show that CSM is able to provide strong consistency, with requests for parking spots within 500 meters served almost always within a second. We extend CSM to allow coherent caching of remote data locally on the mobile phones. This leads to remote requests being serviced up to 48% faster.*

## 1. Introduction

Mobile devices are now ubiquitous. With Moore's Law [5] driving their continuing miniaturization, today's high-end mobile devices have greater computational capability than the desktops available a few years ago. This has led to many computationally intensive tasks, such as video processing, gaming, etc., being performed directly on mobile devices. Yet, mobile devices are still largely used for local computation, with computation based on regional data offloaded to servers in the cloud infrastructure [3].

In recent years, researchers in the sensor and ad-hoc network community (see related work in Section 5) have proposed the harnessing of computational capabilities of a collection of mobile devices, as most location-based services involve data that are inherently distributed, directly sensed and gathered by the mobile devices. If mobile devices can be effectively leveraged for computation on regionally gathered data as well, this can ease the intense bandwidth pressure on the access network infrastructure[1], and lead to faster response times. Since wireless ad-hoc communication consumes significantly less power than cellular networks [24], this will also lead to better battery lifetime of mobile devices.

To realize the above vision, there needs to be a way to effectively program a collection of distributed, mobile devices, where the programmer can reason about and control the entire mobile ad-hoc network (MANET) as a single entity, rather than reason about each node individually. However, these MANETs are very different from static distributed systems since the actual set of participating nodes is dynamic and not known in advance. In order to write practical programs, one has to worry about node mobility, failure and unpredictable message loss patterns. In this paper, we postulate that *consistent shared memory (CSM)* is a key missing piece of the puzzle that will allow programmers to effectively program MANETs, enabling new applications that were not previously possible on such networks. We next present the semantics of CSM, followed by examples of applications that are enabled by CSM over mobile nodes. We then provide an overview of our CSM software layer.

### 1.1 Semantics of Consistent Shared Memory

Parallel programming, where multiple calculations of a single application are carried out simultaneously on multiple processors (which can be housed in different boxes in a data center for large supercomputers, or all on a single chip for recent multicores), has been in use for decades, especially in the scientific community. Exploiting parallelism leads to the improvement of overall application performance. However, writing parallel programs is difficult due to the complexity involved in simultaneously reasoning about multiple threads of execution. In the parallel computing community, a lot of effort has been devoted over the years to come up with efficient parallel programming solutions. One widely accepted technique to ease the programmer's burden is to provide a single memory image, shared consistently across the entire multiprocessor system, so that different processors can be reading and writing to any shared variable and ensured of correct operation. CSM platforms [25] provide programmers with semantics that allow them to reason about parallel program behavior with relatively more ease. Here, we discuss how CSM semantics can also ease mobile, distributed programming.

**Sequential consistency.** In a CSM platform, the reads and writes to memory from each single thread of execution obeys the sequential order that the programmer specifies. Moreover, the reads and writes of the entire parallel program occur in a global sequential order that is consistent across all processors. These semantics allow multiple processors to observe a consistent view of the shared memory, which in turn eases programming effort.

As an illustration, Figure 1 shows a parallel program with two threads of execution running on separate processing agents, each trying to reserve a single shared parking spot. These processing agents could be processors of a multiprocessor machine or in the context of this paper, separate mobile phones. The program is supposed to not allow more than one of the processors to reserve the shared parking spot. In a typical parallel program, there are multiple threads of execution which communicate with each other via shared variables in memory. The car parking program achieves its goal of not allowing more than one car to reserve the parking spot through a shared variable called *spot_owner*, which is initialized to *NULL*. Each thread of execution first performs a read of this

[1]Data access networks of iPhones are already overloaded and running beyond capacity [4] and data providers have started rolling back unlimited data plans [2] due to the costs involved.

variable (say reads R1 and R2) to check whether it is *NULL*, indicating that the spot is available. If yes, the requesting processor is assigned the shared spot, by performing a write to the shared variable (say writes W1 and W2). Global ordering helps the program ensure that each processor first checks whether the spot is available and acquires it only in case it is empty. E.g., if P1's *request_spot* procedure executes first, it will go through the *NULL* check (R1), and spot_owner will be assigned to P1 (W1). Subsequently, when P2's *request_spot* procedure executes the *NULL* check (R2), the *spot_owner* variable will not be NULL, and hence it will not be able to acquire the spot (no W2). Alternatively, if P2's procedure was to execute first, it would acquire the free spot, and P1 will be denied. All processors running this application have to observe the same global order of reads and writes, so that all processors agree on whether it is P1 or P2 which got the shared parking spot. Without consistent global ordering, one processor might think it is P1, while another thinks it is P2 that has acquired the shared spot. Hence, a parking spot may be oversubscribed. This is true as long as at each processor, the reads and writes to the shared variable are done within a *single atomic* execution block. We explain this next.

**Atomic block execution.** Sequential consistency guarantees that all processors observe the same global order of reads and writes to shared memory. In the above example, each processor's write definitely follows the read because of the order in which they appear in the program. Hence, the local order of executions are always (R1 followed by W1) and (R2 followed by W2). However, the global order of execution is non-deterministic. If the global order of the two reads and writes is either (R1, W1, R2, W2) or (R2, W2, R1, W1), then it is guaranteed that the first write to the shared variable is seen by the other processor and the parking spot reserved by only one processor. If, however, the global order of execution is either (R1, R2, W1, W2), (R1, R2, W2, W1), (R2, R1, W1, W2) or (R2, R1, W2, W1), then both processors will assume that they have acquired the parking spot. Thus, sequential consistency alone is not enough to preserve the desired application invariant. Atomic execution is a well understood technique to solve this problem.

Since multiple threads are simultaneously reading and writing to the shared variables, accesses to shared variables have to be performed *atomically*. Atomic execution of a piece of code (e.g., the *request_spot* procedure) implies that the result of execution is such that no read/write from any other thread takes place while this code is being executed. Typically, a read followed by a write to a shared variable is bundled together inside a single atomic block of execution to ensure that the value returned by the read is not modified by another thread before being written. Thus, in the above example, the only allowed global order of executions are (R1, W1, R2, W2) and (R2, W2, R1, W1). Thus, sequential consistency and atomic block execution enable the programmers to write correct CSM programs. Either the hardware or system software of a parallel computer is supposed to provide primitives, like locks, transactional memory, etc., to enable the programmer to control which regions of a parallel program should be executed atomically.

**At-most-once execution**. Shared memory multiprocessors have reliable memory subsystems in which memory reads and writes are guaranteed to be executed *exactly-once*. Their interconnection networks are designed to be fault-free. In contrast, MANETs have a high rate of message losses and it is very difficult for system designers to guarantee reliable message delivery. To deal with this, programmers have to resort to retrying the reads and writes. Managing these retries can be very complex, since even the retries can fail and there is no clear way of knowing what exactly happened in the memory. Thus, in the case of MANETs, *exactly-once* semantics cannot be trivially implemented. Traditional distributed systems face similar problems and resort to a programming semantic

```
Initially: spot_owner = NULL;

P1                               P2
bool request_spot(){             bool request_spot(){
if(spot_owner == NULL)            if(spot_owner == NULL)
{                                 {
   spot_owner = P1;                  spot_owner = P2;
   return true;                       return true;
}                                 }
return false;                     return false;
}                                 }
```

**Figure 1.** A parallel program running on a CSM platform in which two processors, P1 and P2, are trying to reserve a shared parking spot. The program is supposed to not allow more than one of the processors to reserve the shared parking spot.

known as *at-most-once execution* [10] to cope with this problem. This semantic guarantees that a memory operation is executed either once or not at all. Programmers write distributed programs keeping this in mind. We similarly advocate at-most-once semantics for CSM on mobile nodes, as this frees the programmers from having to deal with repeat executions that will wreak havoc on consistency issues, and can be supported at low overheads.

For instance, in the example in Figure 1, let us assume at-most-once execution of the entire procedure *request_spot()*. For now, let us assume that all memory operations to the shared variable are performed at some fixed processing node (called HOME node) and remote nodes communicate their reads and writes via the ad-hoc network to the HOME node. The HOME node executes the memory operations in a sequentially consistent and atomic way, and sends the result back via the ad-hoc network. The CSM layer at the remote nodes retries requests if they do not receive a reply within some fixed time period. The HOME node, however, knows exactly what has happened and whether the procedure has been executed. Hence, it will ensure duplicate requests do not lead to repeat executions of the procedure. If, after a long time, no reply is received by the requester (timeout), the procedure *request_spot()* is assumed to be in an UNKNOWN status, and inferred to have been executed at most once. Thus, even though the remote processor is unsure whether the parking spot is reserved or not, the distributed program still satisfies the invariant that not more than one processor reserves the parking spot. Note that it is possible for a parking spot to be assigned to a processor P1, but P1 does not receive the reply. In such a case, the spot is wasted. However, this does not lead to inconsistency, as not more than one processor grabs the spot. Periodically, wasted spots can be reset by the application [2]. If the fraction of remote shared memory requests that fail and are in UNKNOWN status is quite low (which we observe in our simulation results, presented in Section 4), retries will be rare and not lead to high overheads. Thus, incorporating at-most-once semantics makes CSM an easy-to-program system with limited overheads.

We illustrate how the program of Figure 1 is modified for our CSM software layer (CSMlayer) in Section 2.2. Implementing the program on our CSMlayer enables the application to run over MANETs, while still preserving the semantics of the original program.

## 1.2 Applications Enabled by CSM

Providing CSM over a collection of unreliable mobile devices enables a plethora of applications that were not previously possible. Any distributed application that relies on application data being consistent and reliably updated cannot be currently deployed

[2]For applications that deal with money or increments/decrements, at-most-once semantics can be tweaked to guarantee that the application is off from the actual value by a maximum number of transactions. If strictest consistency is required and at-most-once semantics are not acceptable, then application performance and programmability would have to be sacrificed to ensure that each transaction either completely executes or does not execute at all.

on MANETs. Today, these applications have to be run on a central server in the cloud, with mobile nodes continually accessing the server via some cellular data network like GPRS or 3G. This has delay, bandwidth and power overheads. We next give examples of different kinds of applications that can be enabled by the CSM layer over MANETs.

1. **Distributed applications requiring consistent sharing of data.** Any application that relies on data persistence and shared memory consistency is enabled by our CSMlayer. As an illustration, we demonstrate a car parking application in this paper. We envision a city divided into a grid of multiple regions. Each region has mobile nodes keeping track of the number of free parking spaces in that particular region. A car parking application is deployed on the mobile phones present in the area. The application allows a car from any location in the city to request a parking spot in any region. The car parking application would reply with either a yes, indicating that a parking spot has been reserved for the requester, or a no, indicating that no free parking spot can be reserved right now. At a high level, the application is similar to the implementation shown in Figure 1. The example is for a single parking spot but can be generalized to multiple spots. All updates to the shared variable *spot_owner* should be observed consistently by all nodes. For instance, if P1 does a write to *spot_owner* and it is not reflected at P2 consistently, then a situation can occur in which both P1 and P2 assume that they have reserved the parking spot. The CSMlayer enables such a distributed application to be implemented and deployed on top of a collection of mobile devices. We will illustrate the detailed implementation of the application in Section 2.2.

   Similar to the car parking application, the CSMlayer enables applications in which a set of mobile nodes runs services providing reservations of restaurant spots, hotel rooms, etc., in a geographical area. Essentially, any distributed application that requires consistent reads and writes to shared data can be written atop our CSMlayer.

2. **Distributed applications with high bandwidth demands.** Today, it is possible to deploy the applications highlighted above on a centralized server in the cloud and access data via the cellular data network. Deploying the applications on MANETs would save the expensive cellular data network accesses for applications that consume high amounts of bandwidth. An example of such an application is MicroBlog [17]. It proposes a location-based social platform in which users can share and query content like text comments, ratings, pictures, videos, etc. The authors propose MicroBlog based on an Internet server, but it can be easily extended to a distributed mobile platform. The CSMlayer can be used to meet the consistency demands of this application, while also dramatically reducing the cellular data network bandwidth usage.

   The CSMlayer could benefit an application like a landmark-based route planning application deployed over cell phones as well [31]. In such an application, a car driver or a pedestrian goes to the destination with the route planning application guiding the route based on images of landmarks acquired using cameras and comparing them to a pre-populated database of landmark photographs. If this application is deployed over the CSMlayer, the photos belonging to the local region could be fetched from the locally available cell phones and there would be no need to pre-populate the database of pictures on the cell phones. This would save precious mobile phone disk space if all the photographs were stored locally in advance. If the photographs were being downloaded on demand from a central cloud server, this would save precious cellular data network bandwidth.

3. **Distributed applications requiring a fast response time.** Deploying applications locally over MANETs also leads to dramatically lower response times in the network, as compared to cellular data networks. Safety-critical applications or real-time user response applications can leverage the CSMlayer when cloud deployments are not able to meet the low-latency demands of the application [26]. One can imagine an autonomously driven car which needs to be continually fed with hazardous road situations in order to avoid accidents, such as that studied in [32]. It involves the communication of safety-critical conditions using intra-vehicle communication over a fast, dedicated ad-hoc network like dedicated short-range communication (DSRC). Such applications could code up the critical conditions as shared variables, and use the CSMlayer to maintain them consistently. For instance, shared variables such as coordinates of cars and pedestrians could be maintained consistently in a geographical area to prevent collisions from occurring.

## 1.3 Overview of the Proposed CSMlayer

The CSMlayer aims to ease distributed programming on MANETs by leveraging parallel programming abstractions. The first step towards achieving this goal is to realize the abstraction of multiple mobile nodes in direct communication range with each other as a single stationary processing node. One promising attempt towards this goal is Virtual Nodes [16]. Virtual Nodes abstracts a collection of unreliable mobile nodes into a stationary reliable virtual node (VN). Virtual Nodes is an abstraction and has many implementation proposals. The only practically deployed implementation is described in [11], in which the authors deployed Virtual Nodes on a small set of PDAs. Later, Wu et al. [30] deployed on ns-2 [8] the same Virtual Nodes implementation for larger-scale studies. These implementations allow the Virtual Nodes to be occasionally inconsistent, since their target applications can tolerate this. The CSMlayer semantics do not allow even occasional inconsistencies. To address these inconsistencies, this paper proposes a strongly-consistent Virtual Nodes implementation, called Virtual Nodes-Consistent (VN-C). We describe the details of VN-C in Section 2.1.

Building atop the single, stationary virtual node (VN-C) that governs a small geographical region, we next need a mechanism that connects these stationary nodes together as a grid, enabling sharing of data across an entire city composed of many Virtual Nodes. This paper proposes adding such a CSM software layer (CSMlayer) on top of VN-C, providing a way to consistently access shared data from anywhere, thus enabling new applications and considerably easing the programmer's effort in distributed programming of unreliable mobile nodes. We describe the CSMlayer in Section 2.2. To speed up accesses of data on remote Virtual Nodes, such data are cached on local Virtual Nodes, and we propose Snoopy and Resilient Coherence for the CSMlayer (SRCC), which ensures coherent caching. This improves overall application performance.

Figure 2 gives an overview of the entire CSM architecture. At the bottom are physical mobile nodes that run a state replication protocol and emulate a single immobile consistent VN per small, geographical region (i.e., VN-C). The VN-C layer is responsible for keeping the per-region memory accesses *sequentially consistent* and ensuring *atomic executions* of blocks of code. The CSMlayer runs atop VN-C and allows the application to access shared data belonging to remote regions, via CSM procedure calls. The CSMlayer guarantees *at-most-once* execution of the remote CSM procedures, thereby easing the programmers' burden of worrying

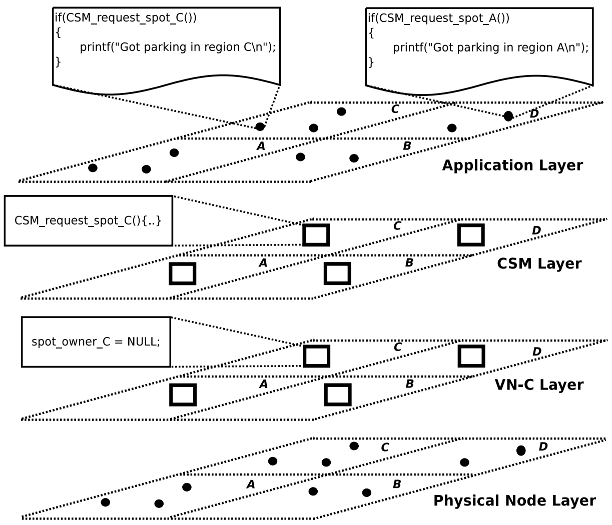


**Figure 2.** CSM architecture. Dots represent mobile nodes while squares denote stationary Virtual Nodes in this area comprising four regions.

```
1. Region Bootup:
   first node enters an empty region;
   broadcast leadership request;
   if receive reply from leader within timeout:
     become non-leader;
   else
     become leader and start VN;

2. Leader Reelection:
   leader periodically broadcasts HEARTBEAT message;
   if non-leader doesn't receive 'n' heartbeats:
     broadcast leadership request;
     if receive reply from leader within timeout:
       remain non-leader;
     else
       become leader with current state;
```

**Figure 3.** Pseudocode for Virtual Nodes

about repeat remote executions. Introducing caches adds consistency challenges, which are tackled by SRCC.

### 1.4 Paper Organization

The rest of the paper is organized as follows. In Section 2, we first discuss the details of Virtual Nodes and identify the sources of inconsistencies present in it. Then, we propose VN-C, which is a consistent implementation of Virtual Nodes. We then describe the CSMlayer in detail. In Section 3, we propose our caching proposal, called SRCC, and describe how it keeps replicated shared data coherent. In Section 4, we describe the implementation details of a toy car parking application that benefits from the strongly-consistent properties of the CSMlayer. We then discuss our experimental setup and present evaluation results showing the effectiveness that the CSMlayer provides for the car parking application. In Section 5, we contrast previous related literature to our work. We finally conclude in Section 6.

## 2. Shared Memory in MANETs

In this section, we start by discussing Virtual Nodes and show how VN-C achieves strong consistency. VN-C handles the *sequential consistency* and *atomic block execution* requirements of our target programming layer. We then present CSMlayer, which allows distributed shared memory programming and implements at-most-once semantics for accessing shared data that are not resident in the local region.

### 2.1 Virtual Nodes-Consistent (VN-C)

In the Virtual Nodes system[3], the entire MANET is divided into equal-sized tiled regions. It is assumed that the mobile nodes have GPS capability so that they can infer the region they belong to. The size of the region is chosen based on the wireless communication range of mobile nodes, such that messages sent from one region can be heard by all nodes in the region, as well as nodes in all neighboring regions. Virtual Nodes offers a programmable *server* in each region. Programmers also write *client* programs to interact with the *servers*. Physical nodes in a region participate in a state replication protocol to emulate a single Virtual Node *server* per region. The physical nodes elect one node as a leader using a simple leader election algorithm. Each node, on entering a new region, sends out a leadership request to all nodes. If the leadership request is not rejected, the node claims itself as the leader and sends out regular HEARTBEAT messages informing other nodes about its leadership. Upon not receiving a certain number of HEARTBEAT messages, a non-leader sends out a leadership request. The pseudocode for the functioning of Virtual Nodes is shown in Figure 3.

The state replication protocol works in a way that client nodes broadcast requests to their local region. The leader as well as non-leaders run the same server application code. All nodes receive client requests and process them according to the application code. However, only the leader node sends response messages. The non-leader nodes buffer response messages until they hear the same response message from the leader. By observing the leader's replies, the non-leaders keep their application state synchronized with that of the leader. On observing a mismatch of its own state from that of the leader, the non-leader synchronizes its state with the leader.

#### *2.1.1 Handling inconsistencies in Virtual Nodes*

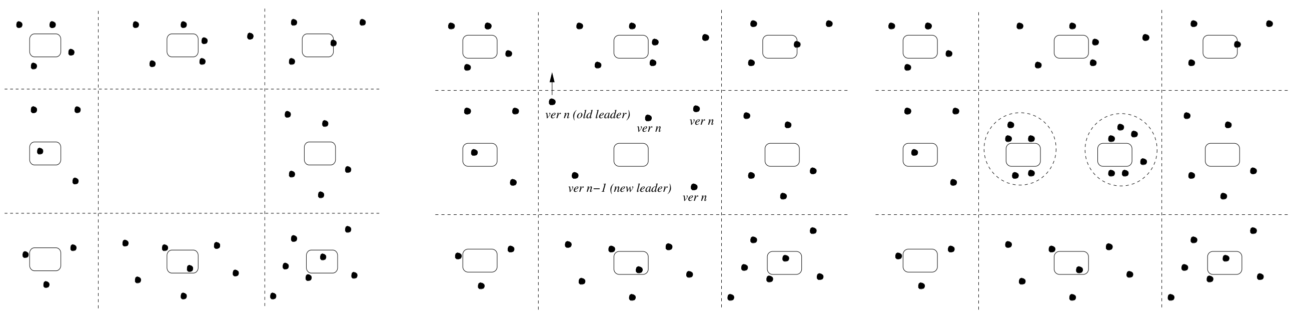
The Virtual Nodes implementation discussed above targets applications that do not require consistent memory accesses and thus the implementation allows occasional inconsistencies:

- **Unpopulated regions.** The mobility of physical nodes following arbitrary motion paths can lead to certain regions in the network being unpopulated for a certain time period, as shown in Figure 4(a). This leads to the application data of the region being lost and when the region reboots, the state is inconsistent with that of the old state.
- **Unsynchronized non-leader becomes leader.** A new leader for a region is elected whenever the current leader leaves the region. The leader election algorithm in the current Virtual Nodes implementation does not guarantee that the new leader elected contains the same application state as the old leader. Thus, a newly elected leader can be inconsistent with the old leader, as shown in Figure 4(b).
- **Multiple leaders in a region.** Excessive congestion in the wireless network can cause high interference, which leads to temporary network partitions. This can lead to a situation in which multiple non-leaders do not hear each other's HEARTBEAT messages and elect themselves as leaders. This situation is shown in Figure 4(c). This results in application data not being in any one consistent state.

To deal with the above inconsistencies present in Virtual Nodes, we redesigned its architecture to be strongly consistent. The pseudocode for the modified Virtual Nodes, called VN-C, is shown in Figure 5. Most of the consistency anomalies in Virtual Nodes occur when the elected new leader is out-of-sync with the old leader. Our VN-C design corrects this and works as described next. (We describe the series of events in a single region, though, without loss of generality, this applies to the entire network of regions.)

**Region boot-up.** Initially, the region is empty. When the first mobile node enters the region, it broadcasts a leadership request. If there is a VN running in this region, the leader replies to the request and the new node becomes a non-leader. The new node waits for the VN reply until a certain timeout, after which it contacts the central server. The central server knows whether a VN

[3]Note that we are assuming the implementation from [11].



(a) Unpopulated regions (b) Unsynchronized leader (c) Multiple leaders

**Figure 4.** Scenarios that create inconsistency in Virtual Nodes. Physical nodes (black circles) emulate stable Virtual Nodes (white rectangles) in each region.

```
1. Region Bootup:
   first node enters an empty region;
   broadcast leadership request;
   if receive reply from leader within timeout:
     become non-leader;
   else
     fetch up-to-date state from central server, become leader and start VN;

2. Leader Reelection:
   current leader leaves the region;
   it broadcasts a LEADER_ELECT message to old region;
   non-leaders send LEADER_REQUEST message along with version # of state;
   old leader elects new leader with up-to-date state and sends back LEADER_ACK;

3. Handling Unpopulated Regions
   if old leader is not able to elect new leader:
     it uploads state to central server;
     next node to enter region fetches state similar to step 1;
```

**Figure 5.** Pseudocode for VN-Consistent

is already running in the region. If the VN is already running, the server simply ignores the leadership request. If not, the most up-to-date shared state of the VN for this region is sent to this node, which starts the VN for this region. This design requires a stable central server that is backed up in cases when there are no physical nodes to emulate and run the VN in a region. The accesses to this central server are assumed to be via always-available cellular data network, like GPRS or 3G.

**Leader reelection.** Until the leader leaves the region, everything runs smoothly with the leader replying to client requests and replicas maintaining backups. When the leader leaves the region, it broadcasts a *LEADER_ELECT* message back to the old region. The replicas receive this message and send back a *LEADER_REQUEST* message to the old leader, along with a version number of the current state of shared data that they have. The old leader receives these requests, selects a replica which has the same version of the shared state as itself, and sends back a *LEADER_ACK* to the selected new leader. If the old leader finds that none of the replicas have the most up-to-date copy of the shared state, it selects any one of the replicas as the new leader and along with the *LEADER_ACK* message, also sends back a copy of the shared state. The above steps ensure that if the region is populated, one replica with an up-to-date state is selected as the new leader.

**Handling unpopulated regions.** The new leader election steps ensure that the inconsistencies described in Figures 4(b) and 4(c) will never arise, since leader election is now coordinated through the single old leader, or the server, rather than all nodes spontaneously electing themselves. However, the scheme still does not address the case when a region becomes unpopulated for a while. We tackle it as follows. If the old leader is unable to find any physical node in the region to elect as the new leader (even after multiple retries), it uploads its shared state to a central server. Later, when a new physical node enters the region, it fetches this state from the central server and starts the VN (as described above). We expect that regions would become unpopulated rather infrequently and thus the accesses to the central server should be minimal, incurring low overhead. This is confirmed from our evaluations (presented in Section 4).

### *2.1.2 Discussion*

We next discuss some of the corner cases that might arise in a practical deployment and how VN-C deals with them.

**Always accessible central server.** In cases when a region's VN is rebooted, our VN-C design relies on the new leader having reliable GPRS/3G connection to a central server. If the server is unreachable before the new leader leaves the region, it is not a problem. This is because we have the node continually send the leadership request until it succeeds. If it does not, then another node that enters the region in the future will try to become the leader. The server ensures no two leaders emerge. Thus, the GPRS/3G connection can be intermittent, but has to be eventually available. Current metropolitan GPRS/3G networks exhibit this behavior and thus VN-C should work well in practical deployments.

**Old leader moves out of range before handing over charge.** If the old leader moves out of range before electing a new leader, the VN relegates to a region reboot scenario. The old leader hands off the up-to-date state to the central server and a new node takes up leadership status and reboots the VN later.

**Handling unexpected node failure.** The VN-C implementation described in this paper tackles message losses and node failures in a region due to mobility. We, however, do not model the battery lifetime of mobile devices and thus do not address cases where a node suddenly dies because of low battery or due to unexpected accidents. In VN-C, physical nodes run a state replication protocol to ensure consistency and only one physical node per region acts as the leader. If any of the non-leaders die unexpectedly, VN-C will not be affected. Only in the corner cases, when the leader has the most up-to-date copy and it suddenly dies, will VN-C run into an inconsistent state. One can address this by having the system software track and notify VN-C when a node is going to run out of battery, or when the user is opting out of the collaborative computing platform. In such cases, the leader can give away its leadership status. We, however, do not model these techniques currently and leave it for future work.

**Actual wireless range smaller than specified range.** Currently, we assume that the old leader is in wireless range of the old region and is entrusted the responsibility of electing the new leader. If the actual wireless range turns out to be much smaller than the specified range, it could lead to many region reboots. To get around this, the old leader can quickly pass its most up-to-date state to the VN running in the region it is currently present in. This region is a neighbor of the old region and can elect a leader for the

```
node's local data
  1 string node_id; (unique id of node)
  2 - - - -
globally shared data
  3 string spot_owner;
  4 - - - -
operations to local data
  5 node_id = "car 1";
  6 - - - -
code to reserve shared spot
  7 bool is_success = CSM_request_spot(node_id);
procedure for shared data access
  8  bool CSM_request_spot(node){
  9  if(spot_owner == NULL)
 10 {
 11    spot_owner = node;
 12    return true;
 13 }
 14 return false;
 15 }
```

**Figure 6.** Pseudocode of the parking spot reservation program written in the CSMlayer API.

old region subsequently. We did not study the modeling of varying the wireless range of mobile devices due to interference and leave the study as future work.

## 2.2 Proposed CSM Software Layer (CSMlayer)

The physical organization of distributed mobile nodes, organized as MANETs, is in some ways similar to that of parallel multiprocessor systems. Like their counterparts, MANETs contain multiple physical processing elements, each with its own associated physical memory. Where MANETs differ distinctly from multiprocessors is that they are connected via an ad-hoc wireless network, whereas multiprocessors are connected using reliable interconnects. Also, the nodes of a MANET are mobile and highly prone to failures, unlike multiprocessors. Building the CSMlayer over VN-C leverages the abstraction of stationary Virtual Nodes, which more closely match fixed multiprocessor nodes. A fixed VN is emulated per region by VN-C and it guarantees consistent access to data maintained by the local region. The VN in each region leverages the CSMlayer to access shared data structures belonging to remote regions. The CSMlayer implements the at-most-once semantics that our target programming layer requires for accessing shared data that are not resident locally at a VN. Next, we describe the CSMlayer implementation in detail.

### *2.2.1 High-level architecture*

Distributed shared memory programming requires a way in which threads of execution on a processor can access memory belonging to remote locations. In the multiprocessor domain, the entire physical memory is shared by all processors and threads running on any processor can access memory from any processor. In MANETs, the communication cost of accessing memory locations belonging to remote nodes is orders of magnitude higher than that in multiprocessors. Thus, in our opinion, it is not appropriate to allow sharing of the entire memory address space. Similar to the partitioned global address space (PGAS) [12] model, we believe that the programmer should be given the flexibility and responsibility to decide which data structures should be shared across the entire MANET and which should be kept private to a single VN. Thus, the programmer has to explicitly reason about which data to share and whenever some shared data are being accessed, the VN explicitly knows about it. Our CSMlayer provides the mechanisms to define shared data and access it.

Figure 6 shows the pseudocode of the parking spot reservation program (illustrated earlier in Figure 1) written over our CSMlayer. The CSMlayer's API requires the programmer to initially specify which data structures should be shared across the entire MANET (lines 3 and 4). The programmer also implements procedure calls that perform memory operations to any of the shared data structures. They are indicated by naming the procedure calls starting with "*CSM_*" (lines 7, 8). All procedure calls that perform operations on shared data structures are handled by the CSMlayer, which ensures that the calls are strongly consistent, atomic, and execute at most once. In the pseudocode, the globally shared variable *spot_owner* is to be kept strongly consistent. Hence, a CSM procedure (*CSM_request_spot*) is used to access the shared variable. This procedure is executed at a VN that is called *HOME* to that shared variable. Each region's VN is responsible for a subset of the entire shared data address space and is called *HOME* to that subset of shared data. This allows the *HOME* of a shared data structure to be determined statically. Since all accesses to this procedure are strongly consistent, it is guaranteed that the shared spot is not occupied by more than one of the processing nodes. This invariant of the application was intended to be preserved, and our CSMlayer enables that.

### *2.2.2 CSMlayer implementation*

The application code interacts with the CSMlayer to maintain its data reliably and consistently. The data that is maintained at the local VN region and operations acting on them are handled locally and the CSMlayer does not interfere with local processing. On a CSM procedure call to access a shared data structure, the CSMlayer is invoked, which deals with all shared data structures and ensures consistent at-most-once accesses to them. A region's local data as well as globally shared data structures are handled by the VN-C replication protocol, which maintains the data reliably in the face of node mobility and packet losses. We next walk through what happens when the application executes a CSM procedure call to access some shared data structure.

1. On CSMlayer invocation, it checks whether the shared data structure belongs to the local region or to a remote region. This depends on which region is *HOME* for the shared data structure. If it belongs to the local region, it simply executes the CSM procedure call on the local VN and sends the reply to the requester. If the data structure belongs to a remote region, the CSMlayer sends a remote request to the HOME region in a hop-by-hop manner via intermediate VNs. The VNs at intermediate regions simply forward the request to the destination HOME region. The CSMlayer at the requesting region logs the request in a pending request queue, until a reply to the request has been received. Each pending request is tagged with a resend timer. If a reply to the pending request is not received by this time, the remote request is retried.
2. At the destination region, the incoming CSM procedure calls invoke the CSMlayer. It executes the remote request in a sequentially consistent and atomic manner and sends out a reply to the original region via intermediate regions. It also logs the reply of each remote request. This is to handle cases when it receives a duplicate remote request that was resent by the requester. To implement at-most-once semantics and not allow repeat executions, the request is not re-executed and the reply that is stored in the log is sent.
3. At the requesting region, the CSMlayer is invoked by the reply. On receiving a reply, it deletes the request from the pending queue and completes the procedure call. It is possible that when the reply reaches the source region, there is no matching request in the pending queue. This could be because a matching reply has already been received at the requesting VN or it has waited for a long time and concluded at-most-once execution and completed the procedure call. In either case, the source region ignores this reply.

### *2.2.3 Discussion*

We next discuss some practicality issues with the CSMlayer implementation described above.

**Atomicity of a CSM procedure in the face of interrupts.** The CSMlayer relies on Virtual Nodes running entire CSM procedures atomically. Primitives, like critical sections, provided by the node hardware and OS are used by VN-C to ensure that entire CSM procedures are executed atomically, even in the face of interrupts, etc. Even in the corner case in which a node is leaving a region while executing a CSM request, the procedure call is executed completely and then the leadership is handed off.

**Finite logging of replies at the HOME region.** In the scheme described above, we log replies of CSM procedure requests at HOME regions. This is done to preserve at-most-once execution semantics and not allow repeat executions of CSM procedure calls. Even though the replies are stored in software, these replies cannot be stored indefinitely. The HOME region can safely throw away CSM requests that have completed (either through the reply, or a timeout). The requester needs to inform the HOME regions about this. We leverage a technique similar to the one used to handle finite write update buffers for the CSMlayer with caching, as described in Section 3.3. Essentially, all CSM requests are incrementally numbered, starting with 1, for every requesting and HOME region pair. Whenever a VN region sends a CSM request to the HOME region, it also sends the highest number of the CSM request it has completed. This allows the HOME region to throw away all CSM replies up to that number, thus enabling logging of finite replies.

**Routing of inter-region messages.** The CSMlayer requires sending of requests and replies from a region to remote regions. We route inter-region requests using the shortest path existent between the regions, similar to the way it was done in [30].

## 3. CSM with Caching

The implementation of the CSMlayer described so far does not include caching of remote data structures. Without caching, all shared memory operations to remote data result in round-trip multi-hop communication between the source and destination regions. In case of message losses, the request/reply is resent. This further adds to communication delay. The communication speed in a wireless ad-hoc medium is very slow relative to computing speed, and thus the delays might be unacceptable, especially for deployments over large geographical areas.

Caching of remote data locally is a well-understood strategy in the multiprocessor domain for tackling this problem. If remote data are cached locally, then there is no need for remote communication and data access is sped up. We propose doing the same in our context, in MANETs. However, caching of data at multiple nodes introduces the problem of keeping the cached copies coherent, i.e., providing applications with the most up-to-date version of shared data. Caching of data complicates the problem of keeping the data access sequentially consistent. For example, in the parking spot reservation application shown in Figure 1, caching leads to multiple copies of the variable *spot_owner* in the system, and CSMlayer needs to ensure a globally agreed-upon order of R1, W1, R2 and W2 by all processing agents. This is an old problem that the parallel computing domain faces and cache coherence protocols are a well understood strategy to addressing this issue.

### 3.1 Cache Coherence on MANETs

Traditional approaches to cache coherence are broadcast-based snoopy protocols [18] and directory-based protocols [21]. Snoopy protocols broadcast data requests so that the remote owner of the shared data can directly reply to the requester. Hence, remote data requests require two logical hops of communication. In a directory-based protocol, in contrast, each remote data request involves an indirection via a centralized node, known as the directory node. The directory node keeps track of all sharer nodes that currently cache a copy of the data, so that it can redirect requests to these remote sharers, which then respond to the requesters. Hence, remote data requests result in three logical hops of communication. One of the reasons for maintaining a directory structure is to avoid broadcasting of requests, so as to save network bandwidth. In the case of MANETs, where the communication cost of an extra logical hop is high and communication is inherently broadcast in nature, we consider snoopy protocols to be a better fit.

Cache coherence protocols come in two flavors: write update and write invalidate. In a write update protocol, whenever a write is performed to shared data, an update containing the modification is sent to all remote caches. The remote caches update their copies on receiving the write update. In a write invalidate protocol, a write triggers an invalidation message to remote caches. These caches mark their copies as invalid and fetch the shared data again at the next access. Write invalidate protocols have the advantage of consuming comparatively lower network bandwidth as write invalidate messages comprise just identifiers of data while write update messages contain the shared data. However, write invalidate protocols cache remote data less frequently. Given that remote accesses in MANETs are slow and significantly cripple application performance, and wireless networks are broadcast in nature, we see write update protocols as more suitable for the CSMlayer.

Multiprocessor cache coherence protocols do not deal with message losses as their interconnection networks are built to be highly reliable. In MANETs, wireless communication is very lossy in nature due to mobility and interference. Thus, a novel cache coherence protocol for MANETs needs to be designed that is resilient to message losses and node failures.

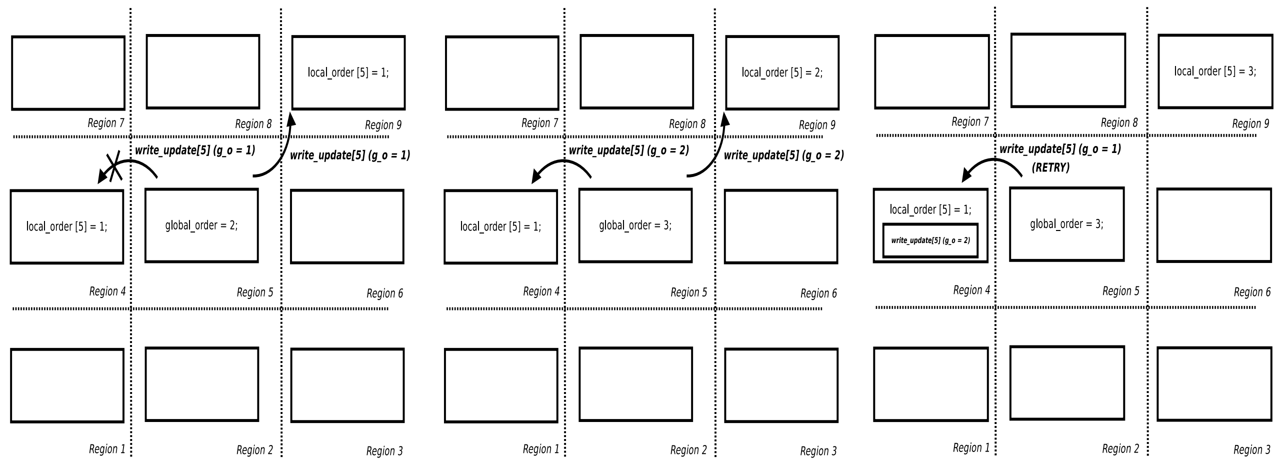
In short, the cache coherence protocol for the CSMlayer should be a write update snoopy protocol that is resilient to message losses and node failures.

### 3.2 Snoopy and Resilient Coherence for CSMlayer (SRCC)

Snoopy cache coherence protocols ensure shared data coherence by forcing all cached copies in the system to see the same **order** of reads and writes. Sequentially consistent execution of reads and writes in distributed networks, where the ordering of packets through the network is non-deterministic, is not trivial. INSO [9] is a multiprocessor snoopy protocol that achieves ordering by assigning ordered numbers to coherence messages and presenting coherence messages to the destination processors in the order of these numbers. Our SRCC protocol is inspired by the ordering technique in INSO. We also use ordered numbers in our cache coherence protocol. We describe our coherence protocol next.

The CSMlayer distributes ownership of shared data across all the regions. Each region's VN is responsible for a subset of the entire shared data address space and is called *HOME* to that subset of shared data. The distribution of shared data structures among various *HOME* regions is done statically. Thus, any node can determine the *HOME* region for a data structure being accessed. The VNs of remote regions cache shared data and our SRCC protocol guarantees that memory operations to shared data resident at any *HOME* region are seen by all remote caches in the same exact order. Observing the same order of updates enables the remote caches to remain coherent. It, however, does not guarantee any strict ordering among memory operations to data from different *HOME* regions. Ensuring the former is sufficient for strong consistency on *a single shared data object* and we believe that trying to guarantee strict ordering among memory operations to all data objects in the entire system is an overkill for MANETs.

To implement strict ordering, each *HOME* region maintains a counter called *global_order*. This counter indicates the number (order) that the next memory operation to shared data of this region will be tagged with. This counter is initialized to 1. Each region in the network also maintains one *local_order* per *HOME* region. This counter indicates which order this local region should



(a) State of system when first write update of region 5 is broadcast. The update does not reach region 4.

(b) State of system when second write update of region 5 is broadcast. This update reaches both the regions.

(c) State of system when first write update of region 5 is retried. Second write update is buffered at region 4.

**Figure 7.** Walkthrough example of SRCC for two writes to region 5. Only regions 4 and 9 are shown for clarity.

accept next. This counter is also initialized to 1. Regions use *local_order* counters to find the order of the next memory operation they should accept. A region accepts a remote memory operation when the *global_order* of the memory update equals the current *local_order* for the corresponding *HOME* region. After accepting the memory update, the *local_order* is incremented. It buffers memory operations with higher orders until their turn arrives. We next walk through how SRCC achieves coherence. Figure 7 shows a walkthrough example of two write updates occurring for shared data that have region 5 as their *HOME* region. For clarity, the figure shows write updates being processed only at regions 4 and 9. Let us assume that the writes are W1 and W2 from Figure 1. To ensure consistency, the cache coherence protocol should guarantee that the write updates are propagated to all caches in the same order. The example will show how updates to this shared data are propagated to remote regions.

1. Figure 7(a) shows how, on the first write to region 5, an update is broadcast to the entire system with *global_order* = 1. The *global_order* is incremented to 2. The write update reaches region 9, but does not reach region 4 due to the message getting dropped. Region 9 accepts the update as it finds the *local_order* = *global_order* = 1. It also increments *local_order* to 2.
2. Figure 7(b) shows what happens on the second write to region 5. An update is broadcast to the entire system with *global_order* = 2. The *global_order* is then incremented to 3. The write update reaches all regions. Region 9 processes the update as it finds the *local_order* = *global_order* = 2. It next increments *local_order* to 3. Region 4 finds that *global_order* does not match the current *local_order*, which is 1. As a result, it buffers the write update.
3. Since region 5's first write update did not reach region 4, it is retried after a while. Figure 7(c) shows this retry reaching region 4 with a *global_order* of 1. Region 4 accepts this update as it finds *local_order* = *global_order* = 1, and increments the *local_order* to 2. It now finds that it has a buffered update with a *global_order* that is 2. Hence, it processes the buffered write update. Thus, write updates are processed in the same order in all regions. This ensures CSM accesses.

## 3.3 Discussion

We next discuss some of the specific properties of SRCC that were glossed over in the earlier explanation.

**Cache size.** Our SRCC protocol allows shared data belonging to remote *HOME* regions to be cached locally at the VN. Although our caches are in software and do not run into tight size constraints, caching a lot of remote data that are not used leads to wastage of mobile device memory, which is limited. In SRCC, we put a user-specified limit on how much remote data is cached and older entries are evicted after the cache size is full. Subsequent accesses to data that are not cached are made from the *HOME* region, similar to how it is done for cold accesses.

**Finite write update buffers at *HOME* regions.** The SRCC protocol requires *HOME* regions to buffer write updates and resend them to regions that have not received the updates. Remote regions tag their *local_orders* when sending remote requests. These *local_orders* essentially work as acknowledgments for write updates seen previously and indicate which remote updates have already been processed by the remote region. Looking at this helps the *HOME* region figure out whether a remote region missed a write update and resend the update, if necessary. Looking at *local_orders* also helps *HOME* regions deduce whether all regions have received a particular write update. If yes, it removes the corresponding update from its write update buffer. This enables the write update buffer to remain small and not run into size limits.

**Finite orders.** The SRCC protocol requires the *HOME* regions to increment the *global_order* counters on every update. The counters are variables in software and on a 32-bit machine the counter can go to a value in billions. After that, there needs to be a mechanism to wrap around and start the counter afresh. We leverage the fact that *HOME* regions are already aware of which *global_orders* have been received by all remote regions. Hence, after a while, the *HOME* regions can start reusing their *global_orders* after informing remote regions.

## 4. Evaluation

To demonstrate the effectiveness of sharing support in distributed mobile nodes, we need to evaluate whether the overheads associated with adding consistency are acceptable. For illustration purposes, we needed a distributed application that relies on the application data being consistent. For our evaluations, we chose a toy car parking application.

## 4.1 Toy Car Parking Application

We model a city as a Manhattan grid divided into multiple VN regions. The car parking application is deployed on mobile devices moving around in cars. Each VN maintains a count of the num-

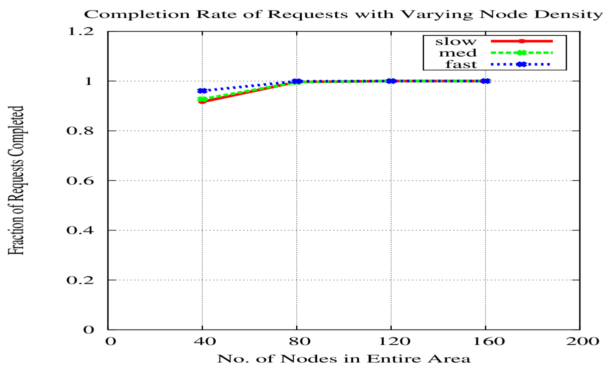


**Figure 8.** Request completion rate for CSM without caching

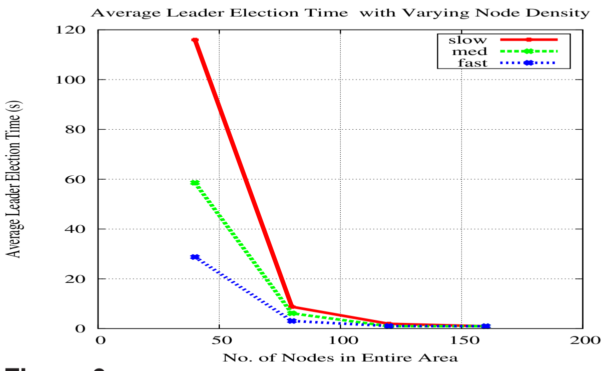


**Figure 9.** Leader election time for CSM without caching

ber of free parking spaces in the region. Cars can request/release car parking spots. The application can be accessed from any part of the city. A car in a particular region of the city can request a parking spot in the same region as well as any remote region in the entire city. In response to the parking spot request, the requesting car gets either a positive or a negative reply. The cars participating in the car parking application are assumed to have capabilities like GPS, ad-hoc wireless communication, and some form of cellular data network connectivity.

Each physical car maintains the application data and runs the application code written on top of the CSMlayer. The application data that need to be maintained consistently are the number of free parking spots in that VN region. A car in a particular region broadcasts a request for a parking spot in some particular region. The VN server running in the original region receives the parking request. If the request is for a parking spot in the local region, the VN server checks for parking spot availability and sends out a reply accordingly. If the parking request is for a spot at a remote region, the parking request is relayed hop-by-hop to the destination region. The destination region checks for the availability of a parking spot and relays back a reply to the requesting region in a hop-by-hop manner. When the reply reaches the original requesting region, the VN server sends the reply to the requesting car.

Next, we present performance and overhead results for our CSM implementation without and with caching. For each of our simulation runs, we verify that CSM access semantics are preserved, by comparing against a simulated centralized server that emulates CSM with at-most-once semantics. Thus, correctness is ensured for our CSMlayer simulations.

## 4.2 Experimental Setup

To evaluate the practical effectiveness of our CSMlayer proposal, we implemented the car parking application on the ns-2.31 [8] simulator. The free-space wireless propagation model was

**Table 1. Settings for motion speed modes**

| | slow | med | fast |
|---|---|---|---|
| Minimum speed (m/s) | 0.73 | 1.46 | 2.92 |
| Maximum speed (m/s) | 2.92 | 5.84 | 11.68 |
| Minimum pause time (s) | 400 | 200 | 100 |
| Maximum pause time (s) | 4000 | 2000 | 1000 |
| Average cross time (s) | 48 | 24 | 12 |

used. The whole area was assumed to be $350m \times 350m$ and was divided into sixteen $87.5m \times 87.5m$ square regions. This approximately corresponds to a five block by five block area in Manhattan. The radio communication range was chosen to be $250m$ and the region size was chosen such that each broadcast message can be heard by all nodes in the same region as well as all neighboring regions. The number of mobile nodes in the entire network was varied from 40, 80, 120 to 160 nodes. Typical car densities in US cities vary from 1700 cars to 8000 cars per square mile [1]. This approximately corresponds to 80-380 cars for our $350m \times 350m$ region. We simulated a density of 40 cars to see its effect on our CSMlayer implementation. Above 120 cars, the regions get dense enough and the CSMlayer shows similar characteristics. Hence, we did not evaluate densities higher than 160 cars. The cars followed the random way-point mobility model. The speeds of the cars were varied as shown in Table 1. Each simulation lasts 40000 seconds.

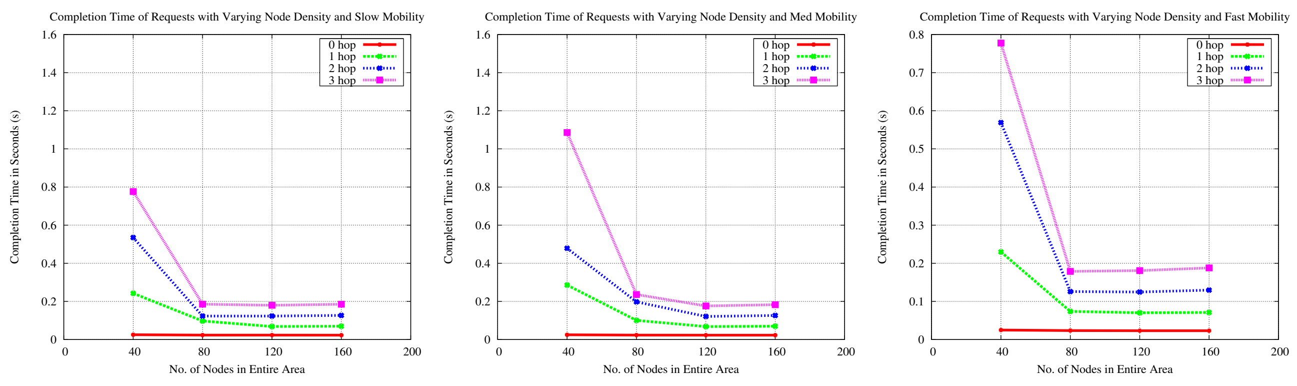
The car parking application is written such that each car entering the city places a request for a parking spot in any of the 16 regions, chosen randomly. The parking spot requests can be either read/write requests. Write requests modify the number of free parking spots, whereas read requests do not. The entire application has 50% read and 50% write parking requests. The closest region is the local region (zero hops) and the farthest is three hops away. Once the request is completed and the reply sent back to the requesting car, the round-trip delay for servicing the request is calculated. This toy application is meant to be representative of distributed programs that share variables across threads and read/write to them concurrently.

## 4.3 Evaluation Results

We next present evaluation results of the toy car parking application, starting with the application written over the CSMlayer without caching at remote regions, followed by an implementation with caching.

### *4.3.1 CSM without caching*

**Request completion rate.** Figure 8 shows the fraction of parking spot requests that complete with varying node density and speeds. In this un-cached CSMlayer implementation, all requests have to travel to the destination region and the reply is relayed back. Since our programming API assumes at-most-once semantics, the requests that do not complete within a timeout are assumed to have been executed at most once (as discussed in Section 1.1). We characterized the maximum round-trip delay of parking requests and chose a conservative timeout period of 5 seconds. The figure shows that when the node density is above a given threshold (80 nodes in the entire area), almost 100% of requests complete. For the experiments with 40 nodes in the entire region, 91.66%, 92.72%, and 96.08% of requests complete for the slow, medium, and fast mobility patterns, respectively. This is despite the requesting car retrying the requests multiple times. The reason why some remote requests never complete is that either the destination or the intermediate regions were inactive for a long period. In such cases, the relayed messages cannot reach their desired destinations. Faster mobility leads to regions getting repopulated much quicker, so that VNs are reinstated more quickly. This can be inferred from Figure 9. It shows the average leader



(a) Slow (b) Medium (c) Fast

**Figure 10.** Request completion time with varying node density and speeds

election time with varying node density and speeds. Leader election time is defined as the time interval between the old leader leaving the region and a new leader being successfully elected. The figure shows that for node densities above 120, the average leader election time is within fraction of a second. Where there are 80 nodes in the entire region, the election time is less than 10 seconds. Thus, there is a very small time window when a region is inactive for high node density cases. When node density gets more sparse, at just 40 nodes across the area, the average leader election time for the slow, medium, and fast cases are 115.96s, 58.57s, and 28.75s, respectively. Such long durations of inactive regions lead to requests not being able to reach their destinations, which, in turn, results in their lack of success.

**Request completion time.** Figure 10 shows the request completion time for successful requests. Request completion time is defined as the time it takes from when a parking spot request is made until the reply reaches the requester. We break down the requests into 0-hop (for local region), 1-hop (for any region that is one hop from the local region), 2-hop (for any region that is two hops from the local region) and 3-hop (for any region that is three hops from the local region) requests. The VN server at the local region maintains a re-sending log for remote region requests. If a reply for a remote region request is not received within a particular timeout period, the request is retried. The request completion time includes the delay caused by the re-sending of requests. The figure shows that for all mobility speeds and node density greater than 120, the request completion time of 3-hop:2-hop:1-hop:0-hop requests is approximately in the ratio 7:5:3:1. Thus, farther the destination region, higher the request completion time. For lower node densities, higher hop requests take even longer than lower hop requests. This is because, when requests are en-route to their destinations, the intermediate or destination regions are found to be inactive and thus the original local region retries the requests. These retries lead to a longer completion time of the request. Clearly, this motivates caching, as caching will reduce these delays to that of a single hop, as we will show later. However, the figures also show that for all configurations, even for 3-hop requests, the request completion times are within 1.1 seconds. Thus, the CSM protocol (even without caching) services requests very quickly and the delays can be acceptable for some applications.

**Central server accesses.** An alternative to CSMlayer is to deploy the application completely on a central server and access the data over the GPRS/3G network. Previous studies [27] show that round-trip latencies over the 3G network vary between hundreds of milliseconds to up to eight seconds when the network is at high loads. The GPRS networks are even slower. The CSMlayer clearly provides a better upper bound on data access latency.

As described in Section 2, our consistency protocol can result in the old leader of a region being unable to elect a new leader. In such a case, it concludes that there are no active nodes in the region and uploads the most up-to-date version of shared data to the central server, via the GPRS/3G network. Data accesses to such networks are expensive and that is one of our motivations for having a distributed ad-hoc network based solution for implementing CSM over mobile nodes. Ideally, we want the accesses to the central server to be as minimal as possible. In Figure 11, we quantify the average leader election failures that occur per region every 10K seconds. On every leader election failure, the central server is accessed twice: once by the old leader to upload the most up-to-date state and then by the new leader to fetch this state. As shown in the figure, for higher node densities (120 nodes and above), the old leader almost always succeeds in electing a new leader. For low node densities (40 and 80 nodes), there are some cases where the old leader is unable to find a node that can take up the leadership role in the region. This happens primarily because the lower node densities lead to some regions being unpopulated for certain periods of time. However, even for lower node densities, such cases are rare and our consistency protocol relies very minimally on central server accesses. For example, the CSMlayer makes fewer than one and seven accesses to the central server every 10K seconds, for the 80-node and 40-node configurations, respectively.

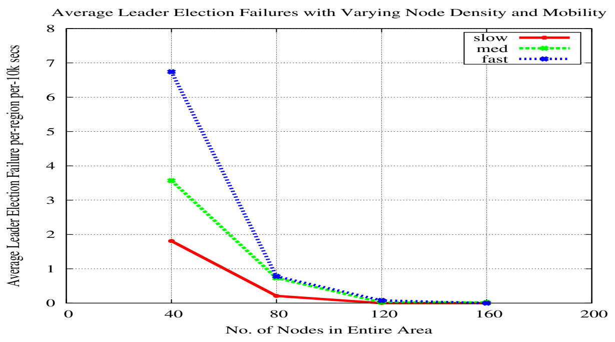


**Figure 11.** Central server accesses

In our simulations, each node requests a parking spot every 100 seconds. Thus, the total number of requests per region in a time period of 10K seconds is 250, 500, 750, and 1000 for 40, 80, 100, and 160 nodes in the area, respectively. Without the CSMlayer, an application requiring consistent access to shared data will have to be implemented on a central server, with all read/write accesses made via the cellular data network. In that implementation, the number of accesses to the central server will be equal to the number of read/write accesses, which is orders of magnitude higher than the central server accesses when using CSM. Thus, the CSMlayer has a huge advantage over the current state-of-the-art in terms of cellular data network bandwidth cost.

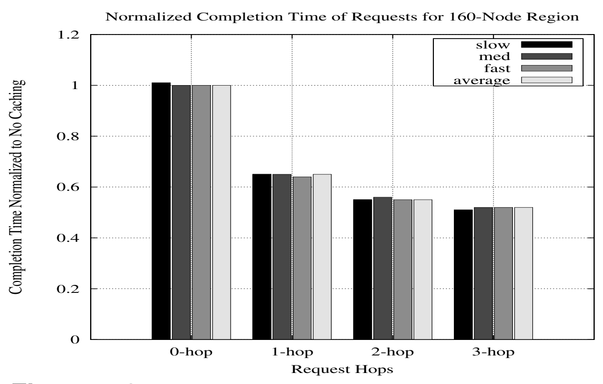


**Figure 12.** Request completion time comparison of caching vs. no caching

### *4.3.2 CSM with caching*

**Request completion time.** Our CSM protocol acknowledges that remote region requests take much longer service times (up to 7×) than local region requests. Thus, our CSM SRCC caching scheme allows the local VN to cache shared data belonging to remote regions, so that applications can get 0-hop request delay for remote data for read accesses[4]. Write accesses still need to travel hop-by-hop to the destination region, trigger coherence actions, and have the reply relayed back. Depending on the fraction of read accesses in the application, the relative advantage of caching remote data would change. Most traditional parallel programs have a majority of read accesses in them, with the percentage of reads sometimes even more than 80% of all accesses [14]. For our evaluations, we chose a modest fraction of 50% read requests and 50% write requests. We performed experiments to evaluate the effectiveness of our caching strategy. Figure 12 shows the request completion time for the different kinds of requests (0-hop to 3-hop) normalized to the case with no caching. The figure shows completion times for varying speeds with the number of nodes in the region being 160. It shows that as the number of hops of the requests increases, the relative performance advantage of caching is greater. On an average, the 1-hop, 2-hop, and 3-hop requests have a 35%, 45%, and 48% lower request completion time as a result of caching. The relative benefits of caching slow down with higher hop count requests. For instance, caching reduces the delay of 2-hop requests by 45% and 3-hop requests by 48%. This is because even with caching, writes still result in multi-hop requests and the delays for 3-hop requests are much higher than that of 2-hop requests. Thus, speeding up only the reads impacts the overall delay to a smaller extent. We present the results only for the case with 160 nodes because we want to isolate the effects of caching from that of VN leadership/membership changes.

**Request completion rate**. We studied the average completion rate of requests in CSM with caching. We found that the fraction of requests that complete for the region density of 40 is 95.6%, 96.32%, and 98.07%, respectively, for the slow, medium, and fast mobility patterns. This is higher than that of CSM without caching, where the corresponding completion rates were 91.66%, 92.72%, and 96.08%, respectively. Caching remote data leads to read requests being satisfied locally, even if intermediate regions or the destination region are inactive. Thus, caching not only helps in the performance of CSM but also in its resilience. Note that for higher node densities, almost 100% of the requests complete even without caching.

[4]Most cache coherence protocols also allow the nodes to locally write to data, if the processing node has relevant permissions. We, however, chose to evaluate a simple design as a first-cut implementation.

**Bandwidth overhead.** Caching of shared data leads to overall performance improvement, but this comes at the cost of increased bandwidth requirement in the ad-hoc network due to the proactive broadcasting of write updates. In our experiments, we analyzed the increase in messages in ad-hoc network due to caching. It should be noted that although write updates add network traffic, they result in reads being satisfied at local regions. This saves extra requests and reply messages. Our experiments show that the CSMlayer with caching has 19% more messages, on an average, as compared to the CSMlayer without caching for node densities 80, 120, and 160 across various speed modes. Interestingly, for the node density of 40 and with varying node speeds, the CSMlayer with caching has 26% fewer messages, on an average, in the network than the CSMlayer without caching. This is because caching saves on the numerous reads requests and replies that fail and are retried in the network with lower densities. Since the actual bandwidth consumption is a function of the application shared state, which varies from application to application, we chose to just measure the message overhead. The write update messages only carry the write that is to be performed on a data structure and should be much smaller than the application shared state, in the usual case. Messages that are used to synchronize the backups with the leader and carry the entire shared state would thus consume considerably more network bandwidth. Hence, the actual bandwidth overhead of caching should be lower than the messaging overhead for real applications.

## 5. Related Work

Our research is related to many programming paradigms developed in the parallel and distributed programming community as well as to programming models proposed in the mobile computing realm. Our work is unique in attempting to practically implement CSM, a programming model which is widely adopted in the parallel programming community[5], on mobile nodes that have not been considered to be suitable for such programming paradigms in the past. We contrast our work with the distributed programming models for MANETs and sensor networks next.

**Distributed programming techniques.** Many researchers have explored programming languages for a collection of resource-constrained devices. Some of the proposals have been for devices that are mobile while others tackle only static resource-constrained nodes. We discuss the most relevant ones here. Hood [29] is a neighborhood programming abstraction for sensor networks in which a node can identify neighboring nodes around it and share its state with them. Hood, however, restricts the sharers to within one wireless broadcast hop and also leaves the consistency, coherence and reliability guarantees to the application level. Similar to Hood, research on Abstract Regions [28] aims at providing abstractions of local neighborhoods for simplifying node-level programming. However, it does not provide any support for consistency or reliability. Kairos [19] is an extension of the python language, which presents an abstraction of a sensor network as a collection of nodes that can all be tasked simultaneously within a single program. Though Kairos enables more effective distributed programming, it does not implement strong consistency and targets only static sensor networks. Pleiades [20] borrows concepts from Kairos and adds consistency support to the programming language. However, Pleiades is primarily targeted towards static sensor networks and is not resilient to node mobility and failures. Our shared memory layer tackles the gaps that exist in the aforementioned works, addressing CSM on mobile nodes with unreliable communications.

[5]Numerous parallel programs have been written using pthreads [7] and OpenMP [6] on CSM machines like Intel Xeon, IBM Regatta, AMD Opteron, etc.

**Virtual Nodes implementations.** The Virtual Nodes [16] abstraction also tackles the programmability of MANETs. The first practical implementation of Virtual Nodes [11] was a small deployment on PDAs. Subsequently, larger-scale simulation studies [30] on ns-2 [8] were also performed. Both these implementations targeted applications that did not require strong consistency guarantees and hence provided abstractions that were susceptible to inconsistent Virtual Nodes behavior. Our shared memory layer, in contrast, provides strongly-consistent shared memory. Certain Virtual Nodes algorithms have tackled and provided some form of consistency support over mobile nodes. Geoquorums [15] provides a quorum-based algorithm to construct consistent atomic memory over Virtual Nodes. However, it is not practically implementable, since it assumes reliable physical-level communication. In [13], Chockler et al. present complex algorithms to implement reliable Virtual Nodes over an unreliable physical network. Their algorithms rely on consensus, which takes multiple rounds of communication to stabilize. Thus, it is very expensive in practice. RAMBO [22] proposes an algorithm to build reconfigurable atomic memory over a highly dynamic system of mobile nodes. However, it does not provide a fully programmable shared memory layer, as the CSMlayer does, and is thus not suited for a general class of applications.

**Cache coherence techniques.** Our CSMlayer allows caching of remote data structures to speed up remote accesses. It proposes and implements a snoopy protocol and reliable cache coherence protocol (SRCC) to keep the multiple cached copies coherent. Implementing snoopy protocols on distributed networks is challenging, as distributed networks do not provide the clean ordering semantics required to implement snoopy protocols. However, there are cache coherence protocols [9,23] in the parallel computing domain that realize snoopy protocols on top of distributed networks. INSO [9] is one such implementation which uses ordered numbers to provide the total ordering semantics that snoopy protocols require. It tags along ordered sequence numbers to coherence requests and processors accept requests based on these ordered numbers. This ensures that all end-points accept requests in the same order. Our SRCC protocol leverages the same idea of using ordered numbers to provide snoopy coherence. However, our SRCC protocol incorporates techniques for resilience to message losses and node failures in MANETs. SRCC provides global ordering only for objects that are homed at the same HOME VN, rather than a total order among all requests in the system. Thus, it maintains ordered numbers per region and not globally. This allows SRCC to avoid the expiration of orders, which INSO requires. In addition, SRCC implements a write update protocol, in contrast to INSO's write invalidate protocol.

## 6. Conclusion

In this paper, we presented a programming model that enables CSM over a collection of mobile devices. This enables an easy to program, distributed mobile platform. We illustrated, using a distributed car parking application, how strongly consistent programs can now be supported and run over distributed mobile nodes. This was impossible earlier. We believe that our CSMlayer is a key enabler of regional computation over a collection of mobile nodes, which leads to lower bandwidth demands over cellular networks, quicker application response times and improved battery lifetime of mobile devices.

## 7. References

[1] A Free Parking Space Grows in Manhattan. `http://ti.org/antiplanner/?p=3565`.

[2] AT&T phases out unlimited data plans. `http://news.yahoo.com/s/ytech_gadg/20100602/tc_ytech_gadg/ytech_gadg_tc2354`.

[3] Cloud computing. `http://en.wikipedia.org/wiki/Cloud_computing`.

[4] Customers angered as iPhones overload AT&T. `http://www.nytimes.com/2009/09/03/technology/companies/03att.html?_r=1`.

[5] Moore's law. `ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf`.

[6] OpenMP. `http://openmp.org`.

[7] POSIX threads. `http://en.wikipedia.org/wiki/POSIX_Threads`.

[8] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[9] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In *Proc. Int. Symp. High Performance Computer Architecture*, Feb. 2009.

[10] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 17(5):3, 1983.

[11] M. Brown, S. Gilbert, N. Lynch, C. Newport, T. Nolte, and M. Spindel. The virtual node layer: A programming abstraction for wireless sensor networks. In *Proc. Int. Wkshp. Wireless Sensor Network Architecture*, 2007.

[12] B. Carlson, T. El-Ghazawi, R. Numrich, and K. Yelick. Programming in the partitioned global address space model. In *Tutorial at Int. Conf. Supercomputing*, 2003.

[13] G. Chockler, S. Gilbert, and N. Lynch. Virtual infrastructure for collision-prone wireless networks. In *Proc. ACM Symp. Principles of Distributed Computing*, 2008.

[14] D. E. Culler, J. P. Singh, and with A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[15] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. L. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distrib. Comput.*, 18(2):125–155, 2005.

[16] S. Dolev, L. Lahiani, S. Gilbert, N. A. Lynch, and T. Nolte. Brief announcement: Virtual stationary automata for mobile networks. In *Proc. ACM Symp. Principles of Distributed Computing*, 2005.

[17] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Micro-blog: Sharing and querying content through mobile phones and social participation. In *Proc. of Int. Conf. Mobile Systems, Applications, and Services*, 2008.

[18] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. Int. Symp. Computer Architecture*, pages 124–131, 1983.

[19] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: A macro-programming system for wireless sensor networks. In *Proc. ACM Symp. Operating Systems Principles*, 2005.

[20] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2007.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. Int. Symp. Computer Architecture*, 1990.

[22] N. A. Lynch and A. A. Shvartsman. RAMBO: a reconfigurable atomic memory service for dynamic networks. In *Proc. Int. Conf. Distributed Computing*, 2002.

[23] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proc. Int. Symp. Computer Architecture*, Jun. 2003.

[24] A. Rahmati and L. Zhong. Context-for-wireless: Context-sensitive energy-efficient wireless data transfer. In *Proc. Int. Conf. Mobile Systems, Applications and Services*, 2007.

[25] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[26] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, 2009.

[27] W. L. Tan, F. Lam, and W. C. Lau. An empirical study on the capacity and performance of 3G networks. *IEEE Trans. Mobile Computing*, 7(6):737–750, 2008.

[28] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. Symp. Networked Systems Design and Implementation*, 2004.

[29] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. Int. Conf. Mobile Systems, Applications, and Services*, 2004.

[30] J. Wu, N. Griffeth, N. Lynch, C. Newport, and R. Droms. Simulating fixed virtual nodes for adapting wireline protocols to MANET. In *Proc. IEEE Int. Symp. Network Computing and Applications*, 2009.

[31] W. Wu, F. Blaicher, J. Yang, T. Seder, and D. Cui. A prototype of landmark-based car navigation using a full-windshield head-up display system. In *Proc. Wkshp. Ambient Media Computing*, 2009.

[32] J. Yin, T. ElBatt, G. Yeung, B. Ryu, S. Habermas, H. Krishnan, and T. Talty. Performance evaluation of safety applications over DSRC vehicular ad hoc networks. In *Proc. ACM Int. Wkshp. Vehicular Ad Hoc Networks*, 2004.