Coherency of Shared Memory in Ad-hoc Networks

Rajeev Alur Michael Greenwald

Department of Computer and Information Science
University of Pennsylvania
Email: alur,mbgreen@cis.upenn.edu

May 21, 2001

Abstract

Memory coherence is a commonly accepted correctness criterion for distributed shared-memory computing platforms. Coherence is formulated assuming a static architecture in which all processors can communicate with one another. In this paper, we argue that the classical notion is not appropriate for ad-hoc networks consisting of mobile devices with constantly changing communication topology. We introduce and formalize a new correctness criterion, called *group coherence*, as a suitable abstract specification for shared-memory computing architectures over ad-hoc networks. We show that two existing systems, the Coda file system and Lampson's global naming scheme, satisfy our definition. Finally, we propose a timestamp-based extension of the popular Snoopy cache coherence protocol for caching in ad-hoc networks, and show it to be group coherent.

1 Introduction

A shared-memory multiprocessor system permits, for reducing memory access latency and interconnection traffic, physically distributed memory storage with potentially multiple cached copies of the same memory location. For ease of programming, it is desirable to hide the details of the caching scheme from the programmers and the protocol must provide programmers with a high-level logical view that all processors access a shared global memory. In particular, in terms of read/write operations for a single memory location, the protocol is required to ensure *memory coherence*. Intuitively, coherence says that sequences of read/write operations local to each processor can be merged to produce a global sequence so that every read operation returns the value most recently written [6, 1, 8]. Coherence can be captured by the following two conditions: (1) *Program Order:* if a processor p writes the value v, then a subsequent *read*-operation by p must return the value v in the absence of any intervening writes by *any* processor; (2) *Write Serialization:* if the set of all values written is $\{v_i | i = 1..\}$, then if *any* processor reads v_j before v_k, then *no* processor may read v_k and later read v_j. The various trade-offs in implementing coherence are well studied, and modern multiprocessor systems employ a variety of sophisticated coherence schemes [8].

Classical specifications of coherence assumes a fixed network architecture in which all processors can communicate with each other. This paradigm is not appropriate to *ad-hoc* networks [9, 5] in which the communication topology is constantly changing. The state of an ad-hoc network can best be modeled as a collection of groups, where each group corresponds to a collection of processes that can communicate with one another. At any point in time, a process belongs to a certain set of groups, and this set changes as a process leaves or joins groups. With the recent popularity of mobile devices, ad-hoc networks are becoming increasingly important.

The question of interest to us is the appropriate high-level abstraction for shared-memory systems over an ad-hoc network. There are at least three motivations for this study. First, it can provide foundations to design architectures that support solving computational tasks over mobile devices in an ad-hoc network [14, 3]. Second, it captures a core problem in configuration management for software distributed on mobile devices where updates are not centralized. For example, consider dynamically downloading Linux kernels and run-time libraries and packages from different mirror sites that are independently updated by independent contributors. The basic problem is determining whether a protocol allowing simultaneous updates and downloads does, or even can, guarantee that eventually the downloaded components come from a consistent set (see also file synchronization [4]). Third, it addresses the question of whether web caches can be built that will not only cache static objects such as html pages and GIF files or even cached responses to dynamic queries, but will also manage shared copies of objects that are modifiable by web browsers and users. In particular, such caches must remain coherent when clients are disconnected or at the far end of a low-power low-speed modem.

Ad-hoc networks offer a fundamentally different service model than classical networks: the *common* case is a collection of mutually disjoint or partially overlapping networks of fully operational processors. The nature of connectivity in ad-hoc networks makes the desired notion of coherence different from the classical one in many ways. First, it seems reasonable to allow incoherent views of the memory across groups, particularly if the groups are disconnected. Second, the value returned by a read-operation cannot be justified by a write-operation if there is no possible way of propagating this value across the dynamically changing groups. Third, a write-operation can be observable within only a selected subset of the groups that can potentially observe this value.

The main contribution of this paper is a definition of *group-coherence* that can be used as a correctness specification for ad-hoc networks. Informally, given local sequences of read/write

operations interspersed with leave/join operations that modify the network topology, we declare them to be group-coherent if we can (1) merge them to create a global sequence, (2) selectively replace leave-operations with reads and join operations with writes, and (3) associate a specific current group for each read-operation and a subset of current groups to broadcast each write operation, so that in the resulting sequence, within each group, every read returns the value most recently broadcast in that group. The specification is liberal enough to allow multiple possible implementations.

We justify our definition in two ways. First, we show that two mature, well-known systems, the Coda distributed file system [11] and Lampson's global naming scheme [13], satisfy our definition of group-coherence. Second, we considered possible ways of extending a simplified version of the popular Snoopy cache coherence protocol [7]. We show a timestamp-based implementation, and establish it to be group-coherent. This exercise also shows the difficulties and challenges of ensuring coherence in ad-hoc networks.

2 Specification of Coherence

In this section, we will assume that there is a single shared object which supports the atomic operations of reading and writing. We use V to denote the values of the shared object, P to denote the set of all processes, and G to denote the set of all groups.

2.1 Classical Coherence

In order to facilitate the extension of coherence to ad-hoc groups, we prefer to formulate the notion of coherence in terms of execution sequences.

A *read/write* sequence is a finite sequence $\sigma = \sigma_0 \ldots \sigma_n$ of symbols, where each symbol σ_i is of the form either $R(p, v)$ or $W(p, v)$, where $p \in P$ is a process, and $v \in V$ is a value. The symbol $R(p, v)$ denotes the event that process p reads the value v, and the symbol $W(p, v)$ denotes the event that process p writes the value v. A read/write sequence σ is *serial* if for all positions i, if σ_i equals $R(p, v)$, then there exists $0 \leq j < i$ such that σ_j equals $W(q, v)$ and for all k, if $j < k < i$ then σ_k is not a write-event. In other words, σ is serial if every read event returns the value of the most recent write [1].

A read/write sequence is coherent if it can be transformed into a serial sequence by shuffling of events belonging to different processes [2]. To be precise, for a sequence σ and a process p, let $\sigma \uparrow p$ denote the sequence obtained from σ by retaining only those events belonging to the process p. For two sequences σ and σ', $\sigma \equiv \sigma'$ holds iff for all processes p, $\sigma \uparrow p = \sigma' \uparrow p$. A read/write sequence is *coherent* if there exists a serial sequence σ' such that $\sigma \equiv \sigma'$.

For instance, consider the sequence

$$\sigma : W(p_1, 5),\ W(p_2, 7),\ R(p_3, 7),\ R(p_3, 5)$$

The sequence is not serial, but is coherent as it is equivalent to the following serial sequence

$$\sigma' : W(p_2, 7),\ R(p_3, 7),\ W(p_1, 5),\ R(p_3, 5)$$

[1] By this definition, every serial sequence must start with a write-event. We can, alternatively, require some initial value to be specified, and allow read-events before the first write to return the initial value.

[2] Events in different processes may be related by a partial order $<_p$. We consider only sequences that do not violate $<_p$, e.g. $i < j \rightarrow \sigma_j \not<_p \sigma_i$.In practice this restricts the set of valid shuffles.

On the other hand, the following sequence is not coherent:

$$W(p_1, 5),\ W(p_2, 7),\ R(p_3, 7),\ R(p_3, 5),\ R(p_4, 5),\ R(p_4, 7)$$

2.2 Group Coherence

While defining the notion of coherence in an ad-hoc network, besides the read/write operations, we need to consider the operations that reflect the changes in the the connectivity among processes. We assume that the basic unit of connectivity is a *group*. A group consists of a set of processes that can communicate among one another. A process can join or leave a group, and can be a member of multiple groups at the same time. Formally, a *(global) history* is a finite sequence $\sigma = \sigma_0 \ldots \sigma_n$ of symbols, where each symbol σ_i is of the form either $R(p, v)$, $W(p, v)$, $L(p, g)$ or $J(p, g)$ where $p \in P$ is a process, $v \in V$ is a value, and $g \in G$ is a group. The events $R(p, v)$ and $W(p, v)$ are the read/write events with the same meaning as before. The symbol $J(p, g)$ denotes the event that the process p joins the group g, and the symbol $L(p, g)$ denotes the event that the process p leaves the group g.

For a global history σ, and a process p, let $B(\sigma, p)$ denote the finite set of groups that p belongs to after the occurrence of events in σ. More formally, let $B(\varepsilon, p)$ be empty, and for $\tau = \sigma \cdot e$, if $e = L(p, g)$ then $B(\tau, p) = B(\sigma, p) \setminus \{g\}$, if $e = J(p, g)$ then $B(\tau, p) = B(\sigma, p) \cup \{g\}$, and $B(\tau, p) = B(\sigma, p)$ otherwise. The history σ is said to be *well-formed* if whenever $\sigma_{i+1} = L(p, g)$, $g \in B(\sigma_{0\ldots i}, p)$. That is, a process leaves only a group that it belongs to. Henceforth, we will assume that histories are well-formed [3]. Analogously, for a global history σ, and a process p, let $C(\sigma, p)$ denote the finite set of groups that p can communicate with after the occurrence of events in σ. A process p can communicate with a group that it belongs to, but also with all the groups that processes in these groups belong to. That is, $C(\sigma, p)$ is the smallest set of groups such that (1) $C(\sigma, p)$ is a superset of $B(\sigma, p)$, and (2) if $C(\sigma, p) \cap B(\sigma, q)$ is non-empty for a process q, then $C(\sigma, p)$ is a superset of $B(\sigma, q)$.

Intuitively, we would like coherence within each group. Thus, a read-operation by a process p can be justified by a preceding write-operation by process q only if p and q belong to the same group, or if the value written by q is read by some other process p' in a group that q belongs to, and p' joins a group that p belongs to (or by similar and longer chains of communication). On the other hand, operations in disconnected groups do not have to be coherent.

Before we formalize the notion of group coherence, we extend the definition of read/write sequences and serial correctness, by considering read/write operations explicitly tagged with the relevant groups. A *group read/write* sequence is a finite sequence $\sigma = \sigma_0 \ldots \sigma_n$ of symbols, where each symbol σ_i is of the form either $R(p, v, g)$ or $W(p, v, H)$, where $p \in P$ is a process, $v \in V$ is a value, $g \in G$ is a group, and $H \subset G$ is a finite set of groups. The symbol $R(p, v, g)$ denotes the event that process p reads the value v from the group g, and the symbol $W(p, v, H)$ denotes the event that process p writes the value v to all the groups in H. Given a group read-write sequence σ and a group g, we use $\sigma \uparrow g$ to denote the read/write sequence obtained by retaining only the events relevant to the group g. That is, for each symbol $\sigma_i = R(p, v, h)$, if $h = g$, we replace σ_i by $R(p, v)$, and if $h \neq g$, we delete σ_i, and similarly, for each symbol $\sigma_i = W(p, v, H)$, if $g \in H$, we replace σ_i by $W(p, v)$, and if $g \notin H$, we delete σ_i. The group read/write sequence σ is *group-serial* if for every group g, the projection $\sigma \uparrow g$ is serial.

We will declare a history to be correct if it can be transformed into a group-serial sequence in two steps. The first step allows shuffling of events belonging to different processes as in the case

[3] By this definition, every process must join one or more groups at the beginning to have meaningful read/write operations. We can, alternatively, require each process to be initially part of a nonempty set of groups.

of classical coherence. To be precise, for a history σ and a process p, let $\sigma \uparrow p$ denote the sequence obtained from σ by retaining only those events belonging to the process p. For two histories σ and σ', $\sigma \equiv \sigma'$ holds iff for all processes p, $\sigma \uparrow p = \sigma' \uparrow p$. The second step transforms a history into a group read/write sequence. Every read-operation $R(p, v)$ is replaced by $R(p, v, g)$ where g is some group that p belongs to at the time of this read-operation. Intuitively, this says that a read can be justified according to any of the relevant groups. Every write-operation $W(p, v)$ is replaced by $W(p, v, H)$ where H is a subset of the groups that p can (transitively) communicate with at the time of this write-operation. Intuitively, this says that a write can be, possibly with the help of other processes, broadcast to a selected subset of the relevant groups. A leave-operation $L(p, g)$ is either deleted, or is replaced by a read-operation $R(p, v, g)$ where v is any value. This says that when a process leaves a group, it could be recording the value of the shared object from that group. Finally, a join-operation $J(p, g)$ is either deleted or is replaced by a write-operation $W(p, v, \{g\})$ where v is the value most recently read or written by p. This says that when a process joins a group, it may broadcast its current value to that group.

We proceed to formally define the second step of the transformation described above. For a group read/write sequence σ and a process p, we use $K(\sigma, p)$ to denote the value used in the last operation by p. For a history σ and a group read/write sequence τ, we define a relation, $\sigma \Rightarrow \tau$, as follows. For the base case, $\varepsilon \Rightarrow \varepsilon$, that is, the empty sequences are related. Suppose $\sigma \Rightarrow \tau$. We consider four possible cases by which σ can be extended.

- Read event: For every group $g \in B(\sigma, p)$, $\sigma \cdot R(p, v) \Rightarrow \tau \cdot R(p, v, g)$.
- Write event: For every subset H of $C(\sigma, p)$, $\sigma \cdot W(p, v) \Rightarrow \tau \cdot W(p, v, H)$ [4].
- Leave event: $\sigma \cdot L(p, g) \Rightarrow \tau$, and for every value v, $\sigma \cdot L(p, g) \Rightarrow \tau \cdot R(p, v, g)$.
- Join event: $\sigma \cdot J(p, g) \Rightarrow \tau$, and $\sigma \cdot J(p, g) \Rightarrow \tau \cdot W(p, K(\tau, p), \{g\})$.

A history σ is *group-coherent* iff there exists global history σ' and a group read/write sequence τ such that $\sigma \equiv \sigma'$, $\sigma' \Rightarrow \tau$, and τ is group-serial.

As an example, consider the following history:

time	p_1	p_2	p_3
$\downarrow$	$J(p_1, g_1)$	$J(p_2, g_2)$	$J(p_3, g_2)$
	$R(p_1, 5)$	$W(p_2, 5)$	$L(p_3, g_2)$
			$J(p_3, g_1)$

This can be shuffled into

$$J(p_2, g_2), J(p_3, g_2), W(p_2, 5), L(p_3, g_2), J(p_1, g_1), J(p_3, g_1), R(p_1, 5)$$

and then transformed into the following group-serial sequence

$$W(p_2, 5, \{g_2\}), R(p_3, 5, g_2), W(p_3, 5, \{g_1\}), R(p_1, 5, g_1)$$

[4] Note that this definition allows a sequence of communications in which some writes are visible and some are not. For instance, consider the case when process p belongs to a group g and groups g and g' have a common process (which is different from p). The specification allows the write by p to be selectively broadcast only to g' but not g. If this flexibility seems undesirable, then one can use a stronger constraint on H: H should equal $H_0 \cup H_1 \cup \cdots H_k$, where H_0 is a subset of $B(\sigma, p)$, and there exist processes $p_1, \ldots p_k$ such that each p_i belongs to some group in $H_0 \cup \cdots \cup H_{i-1}$ and H_i is a subset of $B(\sigma, p_i)$. This stronger specification requires that p brodcasts the write to the groups in H_0, some process p_1 in one of these groups broadcasts the write to the groups in H_1, and so on.

The following history is also group-coherent:

$$J(p_1, g), J(p_2, g), J(p_3, g), W(p_1, 5), W(p_2, 7), R(p_3, 7), R(p_3, 5), R(p_3, 7), L(p_2, g), J(p_2, g)$$

This is because when process p_2 leaves and joins the group again, it can implicitly write the value 7 again justifying the last read-operation by process p_3. On the other hand, the following sequence is not group-coherent:

$J(p_1, g_1)$	$J(p_2, g_1)$	$J(p_3, g_1)$	$J(p_4, g_2)$
$W(p_1, 5)$	$W(p_2, 7)$	$R(p_3, 7)$	$R(p_4, 7)$
		$R(p_3, 5)$	
		$L(p_3, g_1)$	
		$J(p_3, g_2)$	

The only way to justify the read-operation of p_4 is by replacing $J(p_3, g_2)$ by $W(p_3, 7, \{g_2\})$. This would require that 7 was the last value read by p_3, and hence, replacing $L(p_3, g_1)$ by $R(p_3, 7, g_1)$. However, the sequence 7,5,7 of reads by p_3 in the group g_1 cannot be made serial.

3 Implementations of Coherence

The definition of group coherence that we have just proposed is a property of a particular global history of the read, write, leave, and join operations on a single shared memory object. We can extend the definition of group coherence to distributed systems and protocols in the following natural manner. A distributed system operating over a set of shared objects is group coherent, if for every object x, for every history σ, the projection of σ onto the operations related to object x is group-coherent.

A coherency protocol implements the basic high-level operations R, W, L, and J in terms of more primitive local operations and messages sent between processes. The coherency protocol (the rules governing the interaction of the more primitive operations) restricts the set of allowable histories in two ways. First, the protocol imposes timing and ordering constraints on individual high-level operations in a sequence. Second, the protocol determines the value returned by a read operation.

In this section we show that our definition of group coherency is *reasonable* by examining some distributed systems that provide a notion of system-wide coherency, yet allow independent "groups" to proceed in parallel. We verify that these systems are group coherent. We demonstrate that our definition of group coherency is *discriminating* by showing some difficulties in extending a classic coherency protocol to ad-hoc networks in a manner that guarantees group-coherence.

3.1 Illustrative Distributed Systems

3.1.1 Coda

The Venus cache manager in the Coda distributed file system [11] supports disconnected operation of individual nodes. In Coda, the shared objects are entire files and directories rather than individual memory words. A file modification is viewed as a Write that replaces the entire old contents with an entirely new value. A directory update – file creation, modification, or deletion – is similarly viewed as a Write that rewrites the entire directory contents.

When a laptop disconnects from the network, it Leaves the main group and Joins a group of one. While disconnected, all remote operations are logged and the log is replayed on Join. If there are no collisions (an object was not modified in *both* the main group and the disconnected group), then the file modification or directory update is finalized. If both groups modified the file then

some form of recovery occurs: in the case of files the user is asked to manually recover the file given both sets of modifications. In the case of directories, automatic recovery is possible if *different* files were added, deleted or modified in the same directory.

This system is trivially group coherent. When connected, there is one group and coherency is enforced by the distributed file system. While disconnected, each group has a series of reads and writes. Each read is locally justified by a write. When reconnected, if a write conflict occured, then no Reads are permitted until after manual recovery. Manual recovery will likely modify the contents of the file, and is therefore a write. This write justifies all subsequent reads of the file.

3.1.2 Lampson's Global Name Service

Lampson [13] proposes a global name service implemented as a hierarchy of directories, with each directory approximately replicated by a directory copy (DC) stored on different servers. The set of DCs for a given directory are linked together into a ring.

Updates to the name service consist of a path through the hierarchy, and a value. An update either adds the node at the end of the path to the directory or marks the node absent. The update is time-stamped. Updates with later time-stamps supersede nodes with earlier time-stamps.

Updates are performed at individual servers. Individual updates are passed to replicas randomly. Two directory copies exchanging information can be viewed as a group. Consistency is eventually assured by performing reliable sweeps that collect all updates to all elements of the ring. During a sweep we can view each server as joining the ring's full group.

If we view the hierarchy as a single value, it is clearly *not* group coherent. Consider two groups g_1 and g_2 that start with the same value, v. If update u_1 is applied in g_1, then the hierarchy is $u_1(v)$. If update u_2 is applied in g_2, then the hierarchy in g_2 is $u_2(v)$. If the two groups exchange information then the shared value is now $u_1(u_2(v))$ — but that is not equal to either $u_2(v)$ or $u_1(v)$, so cannot be justified.

However, if we look at each individual path as the name for a single shared object, and check for coherency at this granularity, then it is easy to see that every individual value is justified by some write (update), and that the order of writes at each server is a subsequence of the same sequence of writes — that is, the order determined by the time-stamps.

3.2 Extending a cache coherence protocol to an ad-hoc network

It is natural to wonder whether we can extend a classic cache coherency protocol to an ad-hoc network in such a way that it is group-coherent.

3.2.1 A classic coherency protocol

Table 1 specifies an idealized version of a simple write-invalidate snoopy cache coherence protocol (c.f. [7]). The protocol maintains cache coherence between a set P of processors. Each processor caches every shared object, and all processors communicate through a shared channel. The processors can either Read ($v \leftarrow$R) or Write (W(v)), and can broadcast 3 messages over the channel: Read Request (RR), Write Request (WR), and Value (VA(v)). Recall that we consider only one shared object for the purposes of coherence, and therefore we have not included the object identifier in any operations.

We make the following assumptions to simplify our presentation and analysis:

- The shared communication channel is reliable and ordered.

State	input	response = {communication} state var update
INVALID	R	{Send[RR()], Receive[VA(v)]} val← v, state<-READ, return v
	W(x)	{Send[WR()], Receive[VA(v)]} val← x, state←EXCLUSIVE
	VA(v)	{} val← v, state← READ
READ	R	return val
	W(x)	{Send[WR()], Receive[VA(v)]} val← x, state←EXCLUSIVE
	RR	{Send[VA(val)]},state←READ
	WR	{Send[VA(val)]},state←INVALID
EXCLUSIVE	R	return val
	W(x)	val← x
	RR	{Send[VA(val)]},state←READ
	WR	{Send[VA(val)]},state←INVALID

Table 1: State transition table for classic snoopy coherency protocol

- All operations are atomic: (a) The bus does not support split transactions. For example, no messages can be interposed between a WR or an RR and the VA response. (b) Bus requests preempt, abort, and retry any incomplete local operation, serializing the preempted operation *after* the bus request.

- Entries are never flushed from the cache – the cache controller continues to snoop the bus and react to messages relating to cache lines that are in the INVALID state.

A cache entry is a pair (`state`, `val`). Here, `val` is the locally cached value, and `state` is either INVALID when the cache has no information about the object, or READ when `val` is up-to-date, or EXCLUSIVE when `val` is up-to-date and this processor is the only legal writer. The protocol is designed so that when `state` is READ then other caches are either in the INVALID or READ states; when `state` is EXCLUSIVE then all other caches are in the INVALID state.

Informally, Read simply returns `val` when `state` is READ or EXCLUSIVE. In the INVALID state Read sends an RR and returns the value contained in the VA message it receives. Write(v) simply stores v in `val` when `state` is EXCLUSIVE, and in this case, the processor does *not* update any other processor's caches. If `state` is not EXCLUSIVE then Write first broadcasts a WR invalidating the cache entry in all other processors. If a processor p sees a VA message over the bus while waiting to transmit its own VA message then p aborts its own transmission and does not send a duplicate copy of VA.

This protocol clearly maintains coherency. Program order follows immediately from the implementation of Write. Our assumption of atomic broadcast on the shared channel guarantees write serialization because the order of VA messages on the shared communication channel uniquely defines the same order of Writes observed by all the processors.

3.2.2 An extension of the protocol to multiple groups

In this setting, each process belongs to a set of groups. Each message must be sent to a specific group g. We assume that each group supports an atomic broadcast channel within the group, but there are no atomicity guarantees between copies of a single message sent to multiple groups. This poses a challenge for ensuring write-serialization. We use time-stamps to impose a strict ordering on all the writes. For simplicity, we assume each group has a unique integer group ID. A process creates new timestamps by concatenating a time-value and the group-id as the high order and low order bits, respectively. A process constructs the time-value by using the maximum of (a) the local clock, and (b) 1 greater than the time-value of the maximum timestamp it has seen so far. It is easy to

see that the timestamps impose a strict ordering on all writes. Although monotonically increasing per-process counters are sufficient for correctness, we assume roughly synchronized clocks to make the serialized write-order correspond to the intuitive real-time order. We assume there are no failures and no message losses. We assume that a leave event L(g) only happens voluntarily — and therefore a processor in READ or EXCLUSIVE can send a VA message before leaving, guaranteeing that no group will be entirely INVALID[5].

An obvious extension to the snoopy protocol in the previous section is to satisfy reads by querying every group in the set H of groups a process currently belongs to, for the latest value, and acquiring EXCLUSIVE access in *every* group in H before writing. In order to guarantee coherency we must include a timestamp in every message, and we must use this timestamp to resolve conflicts and maintain write serialization. To store this timestamp, and maintain a list of groups we are members of, we extend cache entry to be a 4-tuple {`state`,`val`,`time`,`groups`}.

Leave and Join events update the component `groups`. If a process Joins a group, then all entries whose `state` is EXCLUSIVE must change their `state` to READ. If a process leaves a group g when `state` is EXCLUSIVE or READ, it must broadcast a VA message to g. This VA message uses `time` rather than generating a new timestamp.

A Read while in the EXCLUSIVE or READ `state` returns `val`. If `state` is INVALID then for each group g in `groups`, Read sends an RR message in g and receives a VA message in reply. The value and timestamp from the VA with the latest timestamp become `val` and `time` respectively. If no VA response contains a timestamp more recent than `time` then (a) the process that invalidated this entry is no longer in contact, and (b) the previous `val` and `time` are the most recent valid information we have. Finally `state` is updated to READ.

When `state` is EXCLUSIVE then Write updates `val`, but not `time`, and exits. If `state` is not yet EXCLUSIVE then Write generates a new timestamp as discussed earlier. For each group g in `groups`, Write must send a WR message to g with the value of `time`. If, in response to the WR message it receives a `val` with a timestamp later than `time`, then `state` becomes READ, `val` takes the value from VA, and Write is aborted and not retried. A RR or `val` from *another* group may similarly preempt Write. If, and only if, Write succeeds in sending WR in *all* groups without being preempted by an incoming VA, WR, or RR message then it changes `state` to EXCLUSIVE.

For Read and Write the request-response pairs are atomic within each individual group, however messages may arrive from *other* groups even while waiting for a response to a request. If a process p receives an RR message from a group g, it responds (if `state` is not INVALID) with VA(`val`,`time`)[6]. If VA or WR messages arrive, then the process compares timestamps. If the timestamp of the message predates `time` in the local cache entry, then it treats the message locally as a no-op, and responds with a VA(`val`,`time`). If the timestamp is not earlier than the local `time`, then WR sets `state` to INVALID — note that the process still maintains the old value of `val` and `time`. If `state` was not already INVALID, then the process responds with a VA(`val`,`time`). If the new message is VA, then `state` is changed to READ, `val` and `time` take the value and the timestamp from VA, and it rebroadcasts the VA message on every group in `groups` *except* for the group it arrived on.

If the process is the only member of a group g, then it is possible that no VA message will be forthcoming in response to a WR message. A timeout is used to detect such a case, and `state` becomes AcquireWrite. Note that this can happen in only one case: a W(x) while in the READ state.

[5]Such assumptions are unrealistically optimistic if our goal were to design a *practical* snoopy cache coherency algorithm for ad-hoc networks – but the primary goal of this presentation is to illuminate our definition of group coherence.

[6]As before, if someone else responds with VA first, p does not respond.

State	input	response = {communication} state var update
INVALID	J(g)	{} groups $\leftarrow$ groups $\cup \{g\}$
	L(g)	{} groups $\leftarrow$ groups $- \{g\}$
	R	{$\forall g \in$ groups, Send[RR(g)], Receive[VA(g, x_g, t_g)]}
		Choose max t_g; **if** ($t_g >$ time) **then** (val$\leftarrow x_g$; time$\leftarrow t_g$;) **endif**
		State$\leftarrow$READ, **return** val
	W(x)	t_{new} = GetNewTimestamp();
		{$\forall g \in$ groups :Send[WR(g, t_{new})],Receive[VA(g, x_g, t_g)]
		if ($t_g > t_{\text{new}}$) **then** val$\leftarrow x_g$; time $\leftarrow t_g$; state $\leftarrow$ INVALID; **Abort** W(x);
		else state $\leftarrow$ AcquireWrite; time $\leftarrow t_{\text{new}}$; val $\leftarrow x$; **endif** }
		state $\leftarrow$ EXCLUSIVE;
	RR(g)	--
	WR(g, t)	**if** ($t <$ time) **then**{ Send[VA(g, val, time)]; }**endif**
	VA(g, x, t)	**if** ($t >$ time) **then** val $\leftarrow x$; state $\leftarrow$ READ
		else { Send[VA(g, val, time)]; } **endif**
READ	J(g)	{}, groups $\leftarrow$ groups $\cup \{g\}$
	L(g)	{Send[VA(g, val, time)]}, groups $\leftarrow$ groups $- \{g\}$
	R	**return** val
	W(x)	See W(x) when state= INVALID
	RR(g)	{Send[VA(g, val, time)]}
	WR(g, t)	{Send[VA(g, val, time)]} **if** ($t >$ time) **then** state $\leftarrow$ INVALID; **endif**
	VA(g, x, t)	**if** ($t <$ time) **then** {Send[VA(g, val, time)]}
		else val $\leftarrow x$; time $\leftarrow t$; state $\leftarrow$ READ;
		{$\forall g' \in$ groups, $g' \neq g$:Send[VA(g', val, time)]} **endif**
EXCLUSIVE	J(g)	{}, groups $\leftarrow$ groups $\cup \{g\}$,State$\leftarrow$READ
	L(g)	{Send[VA(g, val, time)]}, groups $\leftarrow$ groups $- \{g\}$,State$\leftarrow$READ
	R	**return** val
	W(x)	val$\leftarrow x$
	RR(g)	{Send[VA(g, val, time)]},state $\leftarrow$READ
	WR(g, t)	{Send[VA(g, val, time)]},**if** ($t >$ time) **then** state $\leftarrow$ INVALID **endif**
	VA(g, x, t)	See VA(g, x, t) when state= READ
AcquireWrite	RR(g)	**Preempt** W(x); {Send[VA(g, val, time)]} state $\leftarrow$ READ;
	WR(g, t)	**if** ($t >$ time) **then Preempt** W(x); state $\leftarrow$ INVALID;
		elseif ($t <$ time) **then** { Send[VA(g, val, time)]} **endif**
	VA(g, x', t)	**if** ($t >$ time) **then Preempt** W(x); state $\leftarrow$ READ; val$\leftarrow x'$; **endif**

Table 2: Snoopy cache protocol extended to multiple groups

To understand how this protocol propagates updates between groups, consider the following example. There are two groups: g_1 contains p_0, p_1 and p_2. g_2 contains p_4, p_3, and p_2. Suppose p_0 writes v_0 at time 1 and p_4 writes v_1 at time 2. If either p_1, p_2, or p_3 try to Read then all 5 processors will get the value v_1. If p_1 reads then p_0 responds, and p_2 forwards VA($v_0, 1$) over g_2 forcing p_4 to respond with a more recent VA($v_1, 2$), which gets forwarded by p_2 back to g_1, overwriting p_0. If p_3 reads then the VA($v_1, 2$) forwarded by p_2 immediately overwrites p_0. Finally, if p_2 Reads then both p_0 and p_4 respond and p_2 forwards only the latest VA. So *any* Read operation forces all processors to agree.

3.2.3 Establishing group coherence

To establish that the protocol discussed in the last section is group-coherent, we discuss how to transform any valid execution of this protocol into a group-serial history.

Assign timestamps to operations in the history in the following manner. Each W operation in the history is assigned the timestamp of the corresponding WR in the trace. (If a process had

Process Groups	P_0 $\{G_0\}$	P_1 $\{G_0, G_1\}$	P_2 $\{G_1\}$
Op,Msg,State	READ	READ	READ
	READ	INVALID	W(0),WR,EXCLUSIVE
	READ	R(0),RR,READ	VA(0),READ
	W(1),WR,EXCLUSIVE	L(G_1),,INVALID	READ
	EXCLUSIVE	INVALID	READ
	VA(1),READ	R(1),RR,READ	READ
	W(2),WR,EXCLUSIVE	INVALID	READ
	EXCLUSIVE	J(G_1),,INVALID	READ
	EXCLUSIVE	L(G_0),,INVALID	READ
	EXCLUSIVE	R(0),RR,READ	VA(0), READ

Table 3: An incorrect execution of the modified protocol

multiple Writes while `state` remained EXCLUSIVE, then there will be multiple W operations in the same process with the same timestamp). Each R operation in the history is labeled with the timestamp included in the VA message in the trace that established the returned value.

Partially order the history by the timestamp order of the reads and writes. If two operations have the same timestamp, then writes precede reads, and writes by the same process preserve their original ordering

Each R(p_0, v, g_0) at t_0 in the history is either preceded (not necessarily immediately) by a W(p_0, v, g_0) in the history with the same timestamp and same p_0, or else by a VA(p_1, g_1, v) message in the trace with the same timestamp and with both p_0, p_1 in g_1. If the latter, then it is similarly preceded either by a W or a VA message in another group g_2. We collect the set of g_i that establishes a chain of VA messages from the reader back to the original writer that justified the read. Construct such a chain for *each* eventual reader of this write. Let H represent the union of g_i from all such chains.

We now informally argue that a shuffle of the history must exist such that for all such $g_i \in H$, J(p_i, g_i) precedes timestamp t_0, and the L(p_i, g_i) is after timestamp t_0. If such a shuffle exists then transforming the write at t_0 to W(p, x, H) and deleting all L's and J's is sufficient to justify all read operations and therefore the history is group coherent.

To see that J(p_i, g_i) can be shuffled prior to t_0, note that J(p_i, g_i) can be moved arbitrarily earlier as long as it does not precede the previous operation from process p_i. But the previous operation from p_i cannot have a timestamp in the trace that is later than t_0 — if it did, then the VA message which arrived later would have been discarded upon receipt, because of an old timestamp. In the process in which the write occured, J clearly precedes the write — else it would not be legal to write in g_n. A similar argument holds for L(p_i, g_i).

It is worth noting that the group-coherence of the protocol is actually quite sensitive to the details of the protocol. If small details are changed then the protocol is no longer group-coherent. For example, consider relaxing the requirement that timestamps and values be stored even when a cache entry is in the INVALID state. In such a case we can no longer guarantee that the timestamps in each process are monotonically non-decreasing, and we cannot prove that a history exists in which the writer and all readers are all transitively communicating at the time of the write. This is demonstrated by the example of Table 3

In the depicted execution, P_1 (by Reading) and P_3 (by snooping) both witness the Write(0) by P_2 in g_1 and both leave the group. P_1 alone Reads P_0's Write(1) in g_0, and is then invalidated by another Write by P_0. P_1 and P_3 both then join g_2. P_1 is INVALID and P_3 responds to P_1's Read with the *old* value of 0 — however, since P_1 was INVALID, and by assumption INVALID states do not

store timestamp, P_1 can not detect that 0 is older than 1. We see that P_1 reads R(0),R(1), and R(0). There is no way to justify both of the R(0)s and make this history group-coherent.

In contrast, in a protocol in which INVALID states store timestamp, then P_1 rejects the VA(0) from P_3 and restores the invalid 1 to both P_1 and P_3 giving a group coherent history.

4 Conclusions

In this paper, we have introduced the notion of coherence for ad-hoc networks. We have a proposed a formal definition of group-coherence, and established its relevance to existing systems and plausible extensions of classical cache-coherence protocols in multiprocessors architectures.

As far as we know, this concept has not been identified previously. In the past, classical coherence has been accepted as *the* correctness criteria, but two broad approaches have been used to allow concurrent progress on a single shared object. The first approach is to allow non-conflicting simultaneous updates to components of the shared object — recognizing that the granularity of the coherence protocol (e.g. cache line, page, file) may be coarser than the granularity of update (e.g. word, cache-line, page). Relaxed consistency models (c.f. [10]) fall in this category, as well as systems that use commutative, idempotent updates (such as [13]). The second approach is to allow conflicting updates to proceed, but then stop the system upon reconnection to perform some form of recovery, usually manual. Most file systems that support disconnected operation (such as Coda [11]) — as well as systems that solve the more general file synchronization problem [4] — are instances of this approach. We note that these two approaches trivially satisfy group coherency. Much other work on correctness in mobile networks (e.g. [2]) assumes either that disconnection is brief, or that disconnected nodes do not make progress, and do not deal with multiple groups. These assumptions do not seem reasonable given current technological trends.

The issues discussed in this paper naturally lead to many questions that we have not addressed. The snoopy protocol described in Section 3.2 should serve only as a first step towards a practical protocol for supporting shared-memory computation in ad-hoc networks. In particular, the issues of efficiency (e.g., reducing the number of messages, discarding cached values when not needed) and liveness (eventual propagation of the most recently written value) need to be addressed. Furthermore, we have only considered coherence in this paper, but there is a more general problem of ensuring *memory consistency*. Addressing this would require generalizing notions such as sequential consistency [12] for ad-hoc networks. Memory consistency in ad-hoc networks is clearly important – however, it seems premature to tackle the problem of consistency (defining consistent orderings of accesses to *multiple* shared objects) before understanding coherence (defining the behavior of reads and writes to a *single* shared object).

References

[1] S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", *IEEE Computer*, 29(12): 66–76, 1996.

[2] S. Alagar and S. Venkatesan, "Causal Ordering in Distributed Mobile Systems", *IEEE Transactions on Computers*, 46(3): 353–361, 1997.

[3] B. R. Badrinath, A. Acharya, and T. Imielinsky, "Impact of Mobility on Distributed Computations", *ACM Operating Systems Review* 27(2): 15–20, 1993.

[4] S. Balasubramaniam and B.C. Pierce, "What is a file synchronizer?", *4th ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MOBICOM '98)*, pp. 98–108, 1998.

[5] S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations", Internet Request for Comments RFC2501, 1999.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and event ordering in scalable shared-memory multiprocessors". *Proc. of the 17th Annual Int. Symp. on Computer Arch (ISCA)*, pp. 15–26, 1990.

[7] J.R. Goodman. "Using cache memory to reduce processor-memory traffic", *Proc. of the 10th Int. Symp. Computer Arch (ISCA)*, pp. 124–131, 1983.

[8] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 2nd edition.

[9] D.B. Johnson, "Routing in Ad Hoc Networks of Mobile Hosts", *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[10] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed Shared Memory on standard workstations and operating systems", *Proc. of 1994 Winter Usenix Conf.* pp. 115–131, 1994.

[11] J.J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System", *Transactions on Computer Systems* 10(1): 3–25, 1992.

[12] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, C-28(9): 690–691, 1979.

[13] B.W. Lampson, "Designing a Global Name Service", *Proc. of the Fifth ACM Symp. on Principles of Distributed Computing*, pp 1–10, 1986.

[14] E. Pitoura and B. Barghava, "Data Consistency in Intermittently Connected Distributed Systems", *IEEE Transactions on Knowledge and Data Engineering*, 11(6): 896–915, 1999.