Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 19
November 12, 2002

Group Communication

Readings:
Keidar, Dolev. Totally Ordered Broadcast in the Face of Network Partitions
Fekete, Lynch, Shvartsman. Specifying and Using a Partitionable Group Communication Service
Yeger-Lotem, Keidar, Dolev. Dynamic Voting for Consistent Primary Components
De Prisco, Fekete, Lynch, Shvartsman. A Dynamic View-Oriented Group Communication Service

Next: Peer-to-Peer Algorithms: Location Services
*** Chord paper: Stoica et al.
*** Pastry, Hildrun et al.
*** CAN paper, Ratnasamy et al.

1 From Last Time

Rui, finish up Cristian-Schmuck algorithm. Three-round majority-based.
Say something about the WAN algorithm?

2 Introduction

A GC service manages two things: membership and communication.
For membership, it provides its clients with a "view" of the current group—a view identifier from a totally-ordered set, and a set of members.
For communication, it provides multicast to the members of the current view of the group, subject to certain ordering and reliability constraints. Different services satisfy different constraints.
The constraints are formulated as a sort of "menu" of independent constraints, from which one can choose any subset.
Stronger constraints are more expensive to implement.

There are many existing GC systems. They have been used in a variety of real applications, including stock exchange, air-traffic management, and collaborative work.

The theory work in this area followed the systems development work.
It has tried to articulate the key services as precise problems, formulate existing algorithms and design new algorithms to solve these problems, and prove correctness and performance properties of the various algorithms.

Around 1996, when I got interested in this area, the prior work consisted of various systems, notably Isis, Transis, Totem, and a beginning of attempts to specify what the systems guaranteed. The specifications used a variety of formalisms (some quite nonstandard), weren't clear about interfaces, some didn't seem right.

We set about trying to do this better.

My first observation was that each system satisfied many properties, and it wasn't clear which were important.
To establish a firm criterion, we decided to pick a characteristic application of the services with a simple correctness criterion, then work backwards to formulate the properties of the service that are needed by that application.

Identifying the right application took some work. It began with an area exam by Debbie Wallach, which surveyed typical applications. When we discussed these with Ken Birman, we got directed to Dahlia Malkhi of the Transis project, who steered us to totally ordered broadcast and consistent replicated data applications. She also steered us to Idit Keidar and her co-workers, who had designed an interesting algorithm to achieve such properties over GCSs.

We started serious work with Keidar and Dolev's PODC 96 paper, which presents an algorithm, used in Transis, that gives TO broadcast using a "partitionable" GC service (one that allows disjoint views to exist at the same time).
We modeled a variant of the algorithm (VStoTO), the goals to be achieved (TO), and the assumed partitionable service (VS), all as I/O automata.
After lots of iterations, we came up with a consistent set of specs, a complete proof of correctness (safety properties), and a conditional latency analysis, based on stabilization as in Cristian-Schmuck (also as in the earlier paper by Dwork, Lynch, Stockmeyer).
Papers: PODC 97, TOCS.

Myla Archer at NRL used our models and proof in her work on formal proofs of distributed algorithms using PVS. Verified all the invariants formally. Working directly from our models and proofs, with no understanding of the algorithm's behavior.

Showed our work on VStoTO to Ken Birman's group at Cornell, including Robert van Renesse and Jason Hickey.
Used this as an example to show them how they could model and analyze their own algorithms.
Modeled two key layers of their latest system, Ensemble.
Main difference is that they include the "virtual synchrony" property, which we had conveniently neglected (because we didn't need it for what we did and anyway, we didn't understand very well what it meant, at the time).
Found key logical error in the reconfiguration part of the algorithm, which had to be fixed in the system.
Error had been undiscovered by the designers for a long time. In fact, the same error appeared in the reconfiguration algorithm of the predecessor system, Horus.
Fixed the algorithm, proved correct.

We went on to consider other GC-based algorithms and treat them similarly. In particular, Roberto De Prisco's PhD thesis followed earlier work by Yeger-Lotem et al., on an algorithm to implement a "unique primary" GCS over a partitionable GCS.

Also, Roger Khazan developed a new algorithm that yields virtually-synchronous GC and optimizes rounds of communication; this is more scalable than previous algorithms. He did complete automaton models, proofs, latency analysis.

3 Building a Totally-Ordered Broadcast Service over a GCS

Although we started with the Keidar, Dolev PODC 96 paper, I'm following the later Avresky volume version here.
First I'll sketch the algorithm from their paper, then go into more details for our presentation.

3.1 Keidar, Dolev algorithm

3.1.1 Overview

The goal is to achieve Totally Ordered Broadcast:
(Safety) Every node should receive a prefix of a common total ordering of messages, and the order should be consistent with causal order, as determined at the client interface with the TO-Bcast service.

(Liveness) If a "majority component" eventually stabilizes, the TO should include all the messages ever sent by members of this component.

Here, a majority refers to a majority of the members of the entire universe.
Majority could be generalized to quorums.

The underlying GCS provides membership and "reliable broadcast".
The specific properties they require appear in Section 3.1, p. 7-8.
Every message has a timestamp, consistent with the client-causal order.
The GCS delivers messages everywhere in timestamp order, though possibly with gaps.
They insist on virtual synchrony (but we didn't understand why—just allows an improvement in the state-exchange protocol).
And Property 3.5, which is a little technical, essentially says that within each view, everyone gets a prefix of a common total order.
Well, it's a little weaker than that—we strengthened it to say this in our work.

3.1.2 The algorithm

Whenever a client submits a message, the algorithm multicasts it using the GCS.
Members acknowledge, also using the GCS.
The messages can be sent around at any time, but the acks are sent in the context of particular views.

The work of totally ordering the messages gets done in majority views only.

Each client message gets assigned a globally-unique timestamp by the GCS, consistent with the client causality order.
The final total order will be consistent with the TS order within each view.
However, the final total order may differ from the timestamp order across views: sometimes later-timestamp messages issued in a majority view will be ordered ahead of (causally concurrent) earlier-timestamp messages issued in a minority view.
This is because the majority view may continue to make progress, accepting and delivering client

messages, while the minority view is disconnected; however, the minority view can continue to accept client messages.

Both the timestamp order and the algorithm-produced order are guaranteed to be consistent with the client causal order.

Each node maintains a local queue MQ, where it stores a tentative ordering of messages.
The queue consists of a prefix of green messages, then yellow messages, then red messages.
The ordering of green messages is permanent, but the others are still undetermined.

In a majority view, messages get ordered as follows (in the normal case, when the view is stable):

When process i receives a message from the GCS, it puts it initially at the end of its MQ, colored red.
When node i acknowledges a red message, in the context of a particular majority view, it colors it yellow.
When i is in a particular majority view and has received acks in that view from all the members of that view for the first yellow message, it colors it green, thus establishing its total order permanently.

Thus, if node i in a majority view v marks a message as green, it must be yellow or green at all members of v.
This will ensure that information about the order propagates to other larger-id majority views: since all majorities intersect, every later majority view will contain some node j that is also in $v.set$.
Node j can convey the ordering information to the later view.

When a new view v forms:
The nodes make their MQs completely identical and completely green before they start processing new messages.
Members of the last majority view preceding v enforce the ordering of green and yellow messages, and everyone puts the red messages at the end, in timestamp order.
Then everything gets colored green.

For more detail, see section 5.5, p. 15-20.

When a new view v forms:

1. Execute Recovery procedure:
   Exchange all state information. Determine the max-previous majority view w.
   All members of w advance their green line to include the most that any of them includes, and retreat the yellow line to include only those that all of them have as yellow.

   Now if $members(v) \subseteq members(w)$ then we're done.
   Else, need to bring the new members of v to a consistent state also:
   A little technical, steps 4-7. Involves some retransmission and adjusting messages and lines to be consistent with some representative member of w. Also, messages from views other than w are ordered in timestamp order at the end, after the green and yellow messages.

2. If v is a majority view, then execute a procedure to Establish the view. A two-round protocol: Mark view v as "attempted", bcast an "attempt" message.
   When attempt messages arrive from all members of v, mark v as "committed", recolor all red messages yellow, and bcast a "commit" message.
   When commit messages arrive from all members of v, mark v as "established" and recolor all messages green.

3.2 Fekete et al. formal treatment

The KD algorithm is very interesting, but I found it a bit informal. It wasn't completely obvious to us that it works, nor even exactly what it guarantees. So, we tried to make this all precise. In the course of doing this, we ended up making the algorithm somewhat more nondeterministic. In fact, the behavior seems to be a bit different, though it certainly preserves the spirit. We do have complete models and proofs, and a decent latency analysis.

3.2.1 Overview

We specify our services as automata, rather than by a list of properties. Advantages: Very clear, concise, systematic, easy to use in formal proofs. Good basis for very formal proofs, e.g., with theorem-provers.
Disadvantages: Properties intertwined in one automaton. Individual properties seem more separable, allow mixing and matching.

This is just for the safety properties. Liveness is replaced by latency bounds, which are expressed as conditional properties, which hold from some point onward. The analysis approach is similar to Cristian, Schmuck (and before that, to Dwork, Lynch, Stockmeyer).

Some of the conditions involve failure behavior, which gets abstracted into special failure status actions.

As in KD, our underlying GCS is partitionable, that is, it allows multiple views to exist at once.
For our application, TO, most of the useful work is accomplished only in majority (primary) views.
We call the GCS VS, but this stands for "view synchrony", not "virtual synchrony", since we don't include virtual synchrony.
Basically, it yields consistent totally-ordered message delivery within each view: everyone gets a prefix of a common total ordering of messages.

In terms of the Chockler et al. survey, we include the properties:
Membership: Self-inclusion, local monotonicity (of views). Our initial view assumption is that there is a particular default initial view, all of whose members start out in that view; everyone else starts out with no view.
We allow partitionable membership.

Communication: Message delivery integrity, no duplication, sending view delivery. Also, safety notification (Rui didn't mention). This is a special output by which the GC service tells its environment that a particular message has been delivered everywhere in the current view.

As before, we use this to implement a single total ordering of all messages.

Diagram, p. 9

3.2.2 Totally Ordered Broadcast specification

TO-machine (for safety) plus TO-property (for performance and fault-tolerance).

p. 10:
Signature: bcast, brcv, to-order

States: global queue, local input pending[p] queues, next[p] pointers to indicate the next message to be delivered.

Transitions: bcast just appends to local queue; to-order incorporates into global queue at some later time; deliveries happen from the global queue, in order.

TO-property, p. 11:
Similar to what we've seen in Cristian-Schmuck.
If a component Q stabilizes, then after at most time b, the delivery service behaves well.
That is, every value sent or received at any location in Q at a time t is delivered to everyone in Q by time $max(t, stabilization - time + b) + d$. Thus, anything sent or received after the end of the transient period takes time at most d to be delivered everywhere.
On the other hand, anything sent or received within this set Q before the end of the transient period still gets delivered everywhere, only now, it might not be until d after the end of the transient period.

This spec is written with Q as a parameter. Later, when we use this in a theorem, we will assume that Q contains a majority of the processes.

3.2.3 View-Synchronous GC specification

VS-machine plus VS-property.

p. 13:
Signature: gpsend, gprcv, safe, newview; createview, vs-order

States: created (for views), current-viewid[p], queue[g].
Also, for each p and g, pending, next (delivery pointer), and next-safe (safe message delivery pointer).

The VS service creates and announces new views.
It manages communication per view pretty much the way TO does overall.

Go over transitions. All straightforward.

VS-property: If component Q stabilizes, then within a short transient period b, everyone in Q gets into a single view containing exactly the set Q as members. Moreover, any messages sent among the processes in Q in the final view are delivered everywhere in that view and are safe everywhere, within time d after the end of the transient period.

3.2.4 The VStoTO algorithm

Entire code is on p. 19 (signature and state) and 20 (transitions). Proof goes on for 10 pages after that. Let's just look over the algorithm by means of the code.

In the state: current keeps track of the current view.
status says where the node is in the protocol: normal is when it's operating normally during a view and the other two are used during reconfiguration.
delay is a buffer for incoming client messages (we call these "values" to distinguish them from the GC "messages")
content is sort of like the red messages—keeps track of all labelled values that you've ever heard about.
nextseqno is used to assign local labels to messages (we allow the local client to assign the label rather than the GCS)
buffer is just an outgoing buffer used for messages to send on the GCS
safe-labels keeps track of which labels are known to be safe
order is the global total order (green and yellow messages)
nextconfirm keeps track of which messages in order are safe (green)
nextreport keeps track of which messages have been delivered to the client

So far, all this is for normal processing. The remaining state components are involved with the reconfiguration.
highprimary is used to record the latest known primary (majority or quorum) view
gotstate records the states that are received in a state exchange
safe-exch records which processes' state exchange messages are known to be safe (have been delivered everywhere)

Normal processing:
bcast appends a new message to the delay buffer.
label takes the first message out of the local delay buffer, assigns it a label, puts it in the local "content" set, and gets it ready to send out using the GCS.
gpsend actually sends it on the GCS
gprcv says what's done when it's delivered: add it to content, and, if the current view is primary, also add it to order (make it yellow)

When a safe indication for a message arrives, mark the message as safe and confirm it (make it green).
Deliver confirmed messages in order to the client

A newview input reinitializes many of the state components in preparation for reconfiguration and other activity in the new view.

Recovery:
This begins with a state exchange: send out your content, order, nextconfirm, and highprimary
When you have received states from everyone in the new view, you are ready to update your state

with new information:
nextconfirm is taken to be the max received, which corresponds to advancing the green border
Then if the new view is not primary, we take the new order to be the "shortorder", which is the order of a chosen representative of the highest known primary.
On the other hand, if the new view is primary, then we take the new order to be the "fullorder", which is defined to be the "shortorder" followed by all the remaining elements of the new content, in label order.
After making all these adjustments, the view is said to be "established", and status reverts to "normal".

Finally, the safe(x) inputs, where x is a state being exchanged, tell a process that the states have been received by everyone in the new view.
When this is known to be true, we simply mark all the labels that appear in the collected states as "safe", all at once.
As for ordinary safe notifications for new messages, this enables them to be confirmed and get delivered to the client.

3.2.5 Proof

LTTR. The general structure of the proof is a forward simulation, showing that the composition of the protocol and VS implements TO.

The key invariant is corollary 6.19, p. 27.
It says, suppose that v is a majority (quorum, primary) view, that has been established at all its members.
Suppose that σ is any sequence of labels that appeared as a prefix of "order" at every member of v while it was still in v.
(Thus, σ is part of the order that was known at all nodes during v.)
Then anywhere in the state of the system that contains information about any larger view, σ is still a prefix of the order.

Roughly speaking, this says that order information that has been determined consistently by all members of any view persists in the same order in all later views.

This key lemma is the heart of the proof. We wanted to prove this as usual, by induction, but it turned out to be hard. We had to strengthen the invariant in a rather technical way; see Lemma 6.18.
Lemma 6.18 has the same conclusion, but based on a weaker hypothesis: instead of saying "for every member p of v, p has established v and σ is a prefix of its order", it requires only: "for every p that has already reached a view higher than v, p has established v and σ is a prefix of its order".

The stabilization analysis is much less formal. It is argued in a modular way, using the stabilization property of TO to prove that of VS (for appropriate values of b and d at both levels).
Namely, we assume that VS satisfies $VS - property(b, d, Q)$ for every Q that contains a quorum.
And we prove that TO built on top of this VS using $VStoTO$ satisfies $TO - property(b + d, d, Q)$ for all such Q.

3.3 Discussion

Sensitive to individual failures, needs to hear from everyone in a view, so requires that the GCS reconfigure after any failure. The frequent reconfigurations can be costly. It's not clear what to do about this, though—the TO requirements are strong, do seem to embody difficulties of consensus, which is costly.

Modeling, proof quite feasible.

4 Building a Unique Primary GCS over a Partitionable GCS

4.1 Overview

Now we switch to a different problem. We assume an underlying partitionable GCS like VS, and we want to build a non-partitionable GCS on top of it. The new GCS should produce a totally-ordered sequence of views, consistent with the client-level causal order, such that at most one view is ever active at one time.

Moreover, consecutive views in the total order should have intersecting memberships, to support transfer of information from one to the next.

We could implement such a service on top of VS by adding a filter that accepts only majority views. Since all majority views intersect, this implies that the membership sets of consecutive views intersect.
But that's overkill. We don't need to have a majority of the entire universe to get these total order, exclusiveness, and intersection properties. We can instead try to ensure just that the membership of each view includes a majority of the members of the previous view.

But it is not so straightforward to implement this idea, because of the asynchrony and possibility of failures while forming the new views.

E.g., consider example from the Yeger-Lotem paper, p. 64:
System has initial view v_1 with members $\{a, b, c, d, e\}$.
a, b and c try to form a new view v_2 with members $\{a, b, c\}$.
They exchange messages. a and b succeed in establishing v_2 locally, but c doesn't finish.
a and b form a new view v_3, whose members are $\{a, b\}$, a majority of the members of their current view v_2.
c connects with d and e, and they form a new view v_4, whose membership $\{c, d, e\}$ forms a majority of the membership of the last view any of them knows about, which is v_1.

So, we have to be careful to handle the asynchrony and failures right. We have to cope with uncertainty about what the current view is.

4.2 Yeger-Lotem et al. algorithm

They don't talk about communication, but just membership. We add the communication in our paper, but it doesn't really add much of interest (or complexity) to the ideas in the paper.

Their basic idea is to form new views in multiple stages. Then a node can keep track of views that it has started trying to form, classifying them as "ambiguous" until they are completely formed.
In the example above, before a and b could actually form v_2, c would know about the attempt to form v_2 and have classified it as ambiguous. That could be used to prevent c from forming the view v_4: v_4 does not include a majority of the elements of v_2, which is at least a *possible* previous view.

They do not keep track of all ambiguous views forever; they use a "garbage-collection" mechanism to forget views older than the last one that is known to be completely installed.

They also address the issue of how to choose the views; in particular, they impose a lower bound on the size of the membership sets, in order to have a reasonable chance of being able to achieve the needed intersection properties.

They start with a generally-known default view W_0.

They define a SubQuorum(v,w) predicate, which says essentially that w contains a majority of the members of v.
Well, they have other technical conditions too.
This condition will be tested in their protocol in deciding if a proposed new view w should be allowed to form.
It will be tested with respect to all "ambigous" v back to the last fully established v.

They observe the property that if SubQuorum(v,w) and SubQuorum(v,w') are both true, then w and w' have at least one member in common.
This will imply that if two views try to form around the same time, they will learn about each other (and can sort out the conflict).

When a proposed new view v is suggested by the underlying GCS, the members of v engage in two steps:
1. Broadcast an "info" message containing info about past views, established and ambiguous. Wait for info messages from all members and collect their info. Decide (based on majority intersection with the largest known previous completed view and all subsequent ambiguous views) if it's OK to try to establish the new view. If so, then record the proposed view v as "ambiguous" and continue (else stop here).
2. Broadcast "attempt" messages. Wait to get attempt messages from all members. If (when) these arrive, establish the new view, and discard information about previous views.

Thus, if any process succeeds in establishing the new view, every other member has recorded it as, at least, ambiguous. So if some members detach before they finish establishing the view, they will still be able to take that view into account in future tests.

Resolving ambiguous sessions: An optimization. Suppose a process p has a view w recorded as "ambiguous". If p succeeds in communicating with all members of w, it can find out if any succeeded in establishing it. If so, then p can reclassify w as established and delete all smaller ambiguous sessions. On the other hand, if no one succeeded in establishing it, then p can simply delete it (doesn't need to take it into account any longer).

4.3 De Prisco et al. formal treatment

Our formal treatment presents similar ideas by providing automaton models for the lower-level service, VS, the new higher-level service, which we call DVS (dynamic view-synchrony), and an algorithm, VStoDVS, which uses VS to implement DVS. The algorithm is very similar to that of YKD.
Then we go a bit further, demonstrating that DVS is useful by showing how it can be used to implement TO. That's similar to what we do in our first paper, but now instead of using majorities, we use the dynamic quorum systems. This means that the proof gets a bit harder, but it follows the same outline.

The formal treatment adds in communication, not just membership, but I don't think that's so important.

We handle the component interfaces carefully—not done in YKD.
In particular, for DVS, we have an explicit action "register" by which the application tells DVS that it is ready to begin actual processing in the new view. This means that the application has completed any preliminary work such as state exchange. This "register" signal indicates to DVS that the view has been "established", as in the discussion above of YKD, which allows the DVS algorithm to stop considering intersections with earlier views.

Our models are state machines, so we can do invariant and simulation relation proofs. YKD are operational, proofs are harder.

VS spec is as in our other paper (a bit different in initial conditions from the initial PODC97 version of the other paper).

DVS spec, p. 5:
Differs from VS in that:
The client is notified only of the "primary" views, which are determined using an intersection condition.
And the application explicitly tells the service when it is done with state exchange, using register inputs.

Go over spec.
The interesting action is the dvs-createview.
To create v, it is compared with every other created view w, both larger and smaller, up to the next registered view in both directions.
For each such w, a nonempty intersection (not majority intersection) is required.

The reason we check in both directions is that the algorithm could create views out-of-order, so a larger view's check might not check the needed intersection property with respect to the smaller view at the time the larger view is created. So it must be checked later, when the smaller view is created.

The check is for nonempty intersection only, not majority. Because this is a spec, not an algorithm, and that's actually what we need to ensure continuity for applications like TO.
The actual algorithm, however, will check majority intersection.
Note that the algorithm checks majority intersection with *known views*, as a way of ensuring

nonempty intersection with *all actual views*.

The algorithm is described on p. 6-8.
Note that, among other things, a node keeps track of: amb, a set of ambiguous views, and act, the last view that is known to have been "totally registered", which means that it is registered at all members

DVS-garbage-collect makes sure that the process updates act when it learns that registration has happened everywhere.

The VS-GPRCV(info,v,V) action receives and incorporates new information about other nodes' act and amb.

The DVS-NEWVIEW is the most interesting action.
It checks that info has been received from everyone.
And it checks majority intersection with act and all views in amb.

The key invariants are 5.2 and 5.3, on p. 9.
5.2 asserts majority intersection between any attempted view and the previous "totally registered" view.
5.3 is a corollary of 5.2, which asserts nonempty intersection between any attempted view any any other view back to and including the preceding "totally registered" view.

4.4 Discussion

Interesting technique for reconfiguration. Supports applications like TO and replicated data that require strong consistency constraints. Tolerates a range of failures.
But, reconfiguration is frequent and costly.

In further work (DC), De Prisco et al. considered how to increase the tolerance to transient failures. Instead of just views with membership sets, use more interesting structure: members plus sets of read and write quorums.
In such a system, work can often be done by accessing quorums rather than everyone. Thus, a new view need not be formed every time someone is failed or unavailable.

We demonstrate the feasibility of this approach by showing how the a version of the Attiya, Bar-Noy, Dolev algorithm for implementing atomic read/write registers using read and write quorums can be made to work over such a service.
Basically, ABD works as usual within each configuration, and resynchronizes when a configuration changes.
The reconfiguration does delay reads and writes in progress. (This is in contrast to the newer Rambo algorithm, which allows the reads and writes to continue during reconfiguration.)