Clock Synchronization

Bring: Notes for scribe. Chalk

Today's reading:
***Attiya, Welch book, chapters 6 and 13
And all the other papers listed (the book chapters summarize these)

Next time:
*** Castro, Liskov paper
*** Rodrigues, Castro, Liskov paper

1 Introduction

Now we move to a setting in which a collection of processes communicate with each other via a set of reliable point-to-point channels.
No shared memory, no separation into clients vs. servers.
In this section, we assume that the set of processes is generally known, so, e.g., they can all communicate with each other.

Many distributed algorithms can take advantage of a notion of "time", based on local "clocks" that are globally synchronized.
What does this mean, and how can this be implemented?

Technologies:

1. Counting steps and carrying out some asynchronous protocol. This leads to purely logical time, with no actual connection with real time.
2. Local quartz clock, like in a wristwatch. This is what stand-alone computers have. The clock provides a way of counting of elapsed time intervals, but does not provide a notion of absolute time unless it's set from some external source. These clocks progress at a rate that is very close to that of "real time", but not exact. Drift can occur. When computers are networked, they can produce "adjustments" for their local clocks in order to obtain a more-or-less common notion of time.
3. Broadcast from an external source, e.g., a satellite. The machine has a receiver that receives the broadcast, or some machines receive the broadcast and sends it to others over a network (as in NTP).

I'll start by reviewing results about logical time, then spend most of the time on achieving synchronization based on local quartz clocks.

Logical time: Lamport is covered in basic course. I'll describe vector clocks (by Mattern and Fidge).

Local approximate clocks: From four papers about various algorithms: by Lamport and Melliar-Smith, by Welch and myself, by Dolev, Halpern, Simons, and Strong, and by Srikanth and Toueg. There are many others, e.g., papers by Cristian and by Marzullo come to mind.

The paper by Lundelius and myself gives a simple limitation on how closely it's possible to synchronize clocks.
The Biaz paper continues this direction. (?)
The Dolev, Halpern, and Strong paper, and the Fischer, Lynch, Merritt paper, show that at least $3f + 1$ processes are needed in order to synchronize clocks where f of the participating processes may be acting faulty in a Byzantine way.

The Attiya-Welch book is a good overall source for this topic.

2 Logical time and vector clocks

2.1 Causality

For an execution of a distributed system, define the *causality order* between events:
All the events at each process are totally ordered, in temporal order.
For any message, the send event is ordered before the receive event.
Take the transitive closure.

Cuts:

Cut for an execution:
A set of events that is closed under locally-precedes (at each process)
A cut is *consistent* if it's closed under causality.

A consistent cut is useful as a snapshot of an execution, e.g., for debugging a distributed algorithm; to calculate the total amount of money in a bank.
The global state corresponding to the cut consists of the state of each process at the cut-point, plus all the messages represented by the message edges that are "crossed" by the cut (those for which the send event appears in the cut but the receive event doesn't).

Thus, an algorithm that determines a snapshot by finding a cut has to discover the messages in transit, not just the process states.

In order to determine cuts and otherwise manipulate the causality ordering, it is helpful to label the events with "logical time" values.

2.2 Totally-ordered logical time

I assume that everyone knows about "Lamport time".
Associate a "time" from a totally-ordered set T with each event in a distributed network algorithm, different for all events, and consistent with the causality ordering on events.

That is, increasing at each node, and send time is less than receive time for the same message. Establishes a total ordering of events that is consistent with what every process sees.

Presented in Lamport's famous paper "Time, Clocks, and the Ordering of Events in a Distributed System", which everyone should read if you haven't already.

Lamport's particular totally ordered set T is the set of pairs (n, i), where n is a local counter (natural number) and i is a process identifier, used to break ties.

Applications:

1. Implement replicated state machines (everyone applies all operations in order of the timestamps of their invocation events).
2. Determine consistent cuts, take global snapshots. By fixing some t and letting everyone provide the last state produced by a local event whose timestamp is $\leq t$, we get states for all the processes that result from a consistent cut. To get a complete global snapshot, we also need to know the messages in transit at the point of the cut. Processes need to determine messages sent by t but that arrived after t.

2.3 Partially-ordered logical time

Mattern and Fidge generalize Lamport's notion of logical time: instead of using a totally ordered set T, they use a poset P. Properties required:

1. No two events are assigned the same logical time.
2. Events at each node are strictly increasing according to P.
3. Each send is ordered strictly before its receive, according to P.
4. Don't have infinitely many events ordered in P before any particular event.

Call this a "weak logical time assignment".

In fact, Mattern and Fidge are interested in weak logical time assignments that represent the causality structure of an execution of a distributed algorithm exactly:
The timestamps of two events are ordered with respect to each other in P exactly if the two events are ordered (in the same order) by the causality ordering.

Applications:
Distributed debugging: A log of local executions at all nodes might be observed later, with associated weak logical times, and the p.o. might be used to infer causality relationships among events.

2.4 Vector timestamps

Mattern and Fidge use a particular poset P: the set of *vector timestamps*, aka *vector clocks*.
Each logical time value is a vector of natural number counts, one position per process in the global set of processes.

(This assumes that we know the set of processes.)
Partially order the vectors componentwise: $V \leq W$ provided that $V[i] \leq W[i]$ for all i.

Their method of assigning elements of P to events is as follows:
Each process keeps a local vector $vector\text{-}clock_i$, where $vector\text{-}clock_i[j]$ contains the latest count i knows for j.
When an event occurs at process i itself, i increments its local count, which is kept in $vector\text{-}clock_i[i]$.
The new value of $vector\text{-}clock_i$ is the logical time assigned to the event.

When i sends, it attaches the new value of $vector\text{-}clock_i$ to the message as a timestamp.
This becomes the logical time of the send event.

When i receives a message, it assigns the receive event a new logical time vector, which is obtained by taking the component-wise max of its current *vector-clock* vector and the incoming vector, and adding 1 to $clock_i[i]$.

Thus, $vector\text{-}clock_i$ contains the latest count i knows for each process.
The key theorems about this logical time assignment are:

Theorem 1: This method produces a weak logical time assignment, as defined earlier.
(In particular, if event e is causally related to event e', then the logical times are ordered, in the same order.)

Theorem 2: If the timestamps of two events in an execution are ordered by this method, then the events are causally related.

Proofs LTTR.

They then show a connection between vector timestamps and consistent cuts.
This provides a criterion for determining if a given cut is consistent, based on examining the logical times of the latest events at all the nodes (latest in the cut set).
That's not the same as determining a consistent cut of a running algorithm, but it does allow us to determine whether a cut is consistent or not.

Theorem 3: Suppose C is a cut of an execution, and let V_i be the vector clock assigned to the latest event at node i in C.
Then C is consistent if and only if $V_j[i] \leq V_i[i]$ for all i, j.

This rules out someone else, say j, knowing more about what i has done than what i is representing in its own cut.
In particular, it rules out j receiving a message from i that i sent after the cut-point at i (so the send event isn't included in the cut at i), and including the receive event in its own cut.

Mattern continues by outlining an algorithm that actually determines a consistent cut.
Basically, it involves the processes agreeing on a future vector timestamp V and then collecting all the events with logical times $\leq V$.
Since this set is closed under causality, it yields a consistent cut.

Again, if we want a global snapshot, we have to augment this algorithm with a scheme to determine the messages in transit at the cut-point, that is, sent by a process before its local cut-point but received after the recipient's cut-point.

This summary has come mainly from the first Mattern paper.
Fidge has similar ideas, but also considers a few other issues, such as a mixture of synchronous and asynchronous communiation, and dynamic process creation and termination.

I didn't find the second Mattern paper that useful.

3 Clock synchronization, no-failure case

3.1 Overview

This comes from the Lundelius-Lynch paper on "An Upper and Lower Bound for Clock Synchronization".

Basically, it defines a simple version of the clock synch problem, without any failures, gives a very simple (toy) algorithm, and proves a lower bound that demonstrates a sense in which the algorithm is optimal.

The lower bound is interesting not just for the result, but because it illustrates a technique of "time-shifting", which has been used in other lower bound results.

Model: Processes communicating by reliable point-to-point channels.
Assume known lower and upper bounds on message delays $[d-u, d]$.
Everyone can communicate with everyone.

Assume each process i has a built in "hardware clock" HC_i.
Model as a local variable whose value represents a "physical time".
For this work, don't consider drift, but assume HC_i progresses at the same rate as real time.
But different processes' clocks can have different values at the same real time—each is equal to $now + c_i$, where c_i is an (unknown) constant.

Processes can read their HC variables but not modify them. They can use them to schedule steps.

The problem is for all the processes to set local "adjustment" variables adj_i to real values to produce new "logical clocks", $AC_i = HC_i + adj_i$.
The resulting logical clocks should, at all subsequent times, all be within some bound ϵ.
The question is how small ϵ can be, for a given number n of processes and given d and u.

The simple answer is that ϵ can be as small as $u(1-1/n)$.
A trivial 1-round algorithm can achieve that.
But also, this can be shown as a lower bound.

3.2 Algorithm

Everyone reads their local HC variables and instantly sends the value to everyone else.
They each process i sets $diff[j]$ for each other process j, which is i's estimate of the difference $HC_j - HC_i$.
Specifically, add $d - u/2$ (the midpoint value for the interval of possible delivery times) to the incoming value, to estimate the current HC_j, then subtract HC_i.
Set $diff[i] = 0$.

Take the average (mean) of all these differences (including your own), to obtain adj_i.

Claim this achieves synchronization $u(1 - 1/n)$:
Consider two processes i, j.
The difference in their estimates of the clock of some other k is at most u—this comes about, e.g., if the message from k to i goes as quickly as possible and the message from k to j goes as slowly as possible.
Also, the difference in their estimates of the clock of i or j is at most $u/2$—since one has the correct value and the other is at most $u/2$ off.
So, the discrepancy in final logical clocks is at worst the average of these discrepancies, namely, $1/n(2(u/2) + (n - 2)u)$, which is equal to the given bound.

3.3 Lower bound

$u(1 - 1/n)$ is also the best that can be achieved under the given circumstances.

To see this, we use a simple technique of constructing a sequence of alternative executions by simply shifting some of the events with respect to real time.

Assume we have an algorithm A that synchronizes to within ϵ and show that $\epsilon \geq u(1 - 1/n)$.

Start with an execution α with a particular set of hardware clock functions HC_i, and with fixed message delays: always $d - u$ from any process to one with larger index, and always d from any process to one with smaller index.
That is, messages are consistently fast from lower to higher index processes and slow in the opposite direction.
This results in clocks AC_i, which are assumed to be within ϵ of each other.

Now, for any k, one can construct a new execution α_k in which all events at processes $1, ..., k$ are shifted earlier in real time by u.
For this to look the same as α to all the processes, the HC clocks for those processes must also be shifted by u (later).
Now messages from processes with indices $\leq k$ to processes with indices $> k$ take the max time, and messages in the opposite direction the min time.
Messages within each of these two groups still take min time from lower to higher indices and max time from higher to lower indices.

This execution α_k produces the same adjustments as α, but slightly different adjusted clocks, because some of the hardware clocks have shifted.

In particular, for each $i \leq k$, the adjusted clock for process i is $AC_i + u$, whereas for $i > k$, the adjusted clock is just AC_i.

The logical clocks produced by α_k must also satisfy the ϵ synchronization condition. This adds extra constraints.
Claim: $AC_k + u \leq AC_{k+1} + \epsilon$.

So, we have $AC_1 \leq AC_2 + \epsilon - u$,
$\leq AC_3 + 2\epsilon - 2u$
$\leq AC_n + (n-1)\epsilon - (n-1)u$

But we also have $AC_n \leq AC_1 + \epsilon$.

Therefore, $AC_n \leq AC_n + n\epsilon - (n-1)u$, which implies $0 \leq n\epsilon - (n-1)u$, which implies the conclusion.

3.4 Discussion

Not so surprising—uncertainty about message delays imposes limits on how closely clocks can be synchronized, even in the absence of drift and failures.
But at least we know that (without drift and failures) clocks can be synchronized to within the limits imposed by message delay uncertainties.

These results assume worst-case uncertainty about message delays.
Some practical algorithms, e.g., one by Cristian, attempt to take advantage of randomness by making repeated attempts in order to get better estimates.
Cristian uses two-way message exchanges instead of one-way to allow processes to estimate each other's clocks.
He notes that the originator of an exchange can get a better estimate from a fast exchange than a slow exchange.
E.g., an exchange whose round-trip-time measures $d_1 << d$ on the originator's local clock allows the originator to estimate the clock of the other party to within $d_1/2$.

To get practical clock synch algorithms, we need to consider other issues like drift and failures, which we do in the next section.

Extensions of this work to other kinds of network topologies have been made by Halpern, Megiddo, and Munshi, 1985, and by Biaz and Welch, very recently.

4 Clock synchronization with failures

4.1 Overview

A very quick review of a series of clock synch algorithms that appeared in the literature around the same time.

Lamport, Melliar-Smith

Lundelius, Lynch
Srikanth, Toueg
Dolev, Halpern, Simons, Strong

These tolerate various kinds of failures.

4.2 Lamport, Melliar-Smith

4.2.1 Overview

Regarded as seminal paper, because it defined the practical clock synch problem, allowing for drifts and Byzantine failures.

The problem is similar to what I've already defined. Processors have HC_i hardware clocks and set adjustments, adj_i, in order to obtain adjusted logical clocks $AC_i = HC_i + adj_i$.
Now the clocks can drift, which means that they progress at a rate that may vary, but remains close to 1; expressed by saying that the rate remains in the interval $[1 - \rho/2, 1 + \rho/2]$.
Also, some of the processes may exhibit Byzantine failures, which means they can lie about their clock values.

They do not address the problem of achieving initial synchronization, but rather, of maintaining synchronization assuming the clocks are initially synchronized.
Thus, the problem here involves coping with the drift.

Methods for achieving initial synchronization are similar to those for maintaining synchronization. LTTR.

They give three algorithms.
Each is intended to be executed periodically, where the times at which they are executed are determined by the local logical clocks.
That is why they need these clocks to be initially synchronized—the processes have to exchange messages with each other, and need to know when to do this.

4.2.2 Algorithm 1 (Interactive Convergence)

This maintains synchronization to within ϵ.
Each process sends its logical clock value to each process, including itself.
Process i collects these messages and replaces any message containing a clock that differs by more than ϵ from its own clock, with its own clock.
(It must be that processes with "outlying" clocks are faulty.)
Process i then resets its own logical clock to the average of all the received values.

In this algorithm, any two nonfaulty processes i and j have close estimates for any nonfaulty process—the only discepancies are a result of message delay uncertainties.
Also, their estimates for faulty processes differ by at most approximately 3ϵ.
So the difference in the averages is at most $3\epsilon(f/n)$.
Assuming that the total number of processes is at least $3f + 1$, the computed averages differ by

less than ϵ.

They show that ϵ is maintained.
The precise calculations are a bit intricate, because of the approximations.

This algorithm is basically taking averages as in the non-fault-tolerant algorithm discussed earlier.
To make it fault-tolerant, we ignore the outlyers.
The trick is that if you have enough processes, the correct ones' estimates "swamp out" the incorrect ones.
Interesting that we need $3f+1$, which is the same bound as for ordinary Byzantine agreement.

4.2.3 Algorithm 2: Interactive Consistency

Now, at each time when the synchronization algorithm is to be executed, the processes simply run Byzantine agreement on the values of each other's logical clock values.

This requires $3f+1$ processes, because of the BA bounds on processors.
It also takes $f+1$ rounds.

It ensures, however, that all nonfaulty processes obtain approximately equal estimates for *every* process' clock—not just the nonfaulty ones.
Then take the median, which is guaranteed to be somewhere in the "correct" range.

Comparing the two algorithms above, they see that both can maintain approximately the same level of synchronization, and of course both require $3f+1$ processes.
To achieve this level of synchronization, the convergence algorithm has to be executed more frequently—by a factor of $3f+1$.
However, the number of rounds required by the IC algorithm is much more anyway—by a factor of $f+1$.

4.2.4 Algorithm 3: IC with authentication

Similar, but assumes authentication of all messages based on unforgeable digital signatures.
Uses Byzantine agreement algorithm designed for this setting.
These still require $f+1$ rounds.
However, they have no limitation on the number of processes.

Halpern, Simons, and Strong developed an improved authenticated algorithm, LTTR.

4.3 Lundelius, Lynch

This followed the Lamport, Melliar-Smith paper and developed a somewhat nicer and better-performing convergence-style algorithm.

It also shows how to allow failed processes to rejoin the system, and how synchronization can be established in the first place.

The convergence algorithm is similar to LM. But instead of resetting everything that's too far from your own value, simply throw out the f highest and f lowest received values at each round, and average the rest.
Can use different averaging functions—to be specific, use the midpoint of the remaining range of values.

Claim this is a good "fault-tolerant averaging function" because it halves the range of the values of nonfaulty processes when it is applied everywhere.
For some intuition, consider 4 processes, 1 faulty process.
Two processes, i and j, receive three values in common, i receives one lower and j receives one higher than the other three.
They will both throw out the outlyer (so it doesn't matter how big or small it is).
Each will also throw out one of the others. When they take the midpoints of what they have left, the distance between these two midpoints is only half the distance between the smallest and biggest of the three "correct" values.

This lets us claim more convergence for each round than LM, which means that the algorithm can be executed less frequently in order to maintain a reasonable level of synchronization.

Reintegrating a failed process: Just let it monitor the ongoing messages until it "reorients" itself enough to know that it's getting all the messages for a particular synchronization round.
Then it simply sets its clock according to the received messages and rejoins the main algorithm.
While it is in process of rejoining, it doesn't matter whether the other processes consider it to be faulty or nonfaulty—as long as it fits within our allotment of f failures.

Thus, the algorithm can be made to tolerate f "concurrent" failures, not just f over all time.

Another algorithms, similar to the main algorithm, is given for the different task of establishing initial synchronization. This algorithm does interface with the main algorithm, allowing a switch from one to the other when synchronization is close enough—see Welch's thesis.

The algorithm for initial synch is also round-based, and uses the same averaging function.
The main difference is that the rounds can't be triggered by the local clocks reaching certain times. Rather, they are triggered by a combination of elapsed physical time and receipt of special messages.

Slightly complicated, LTTR,

4.4 Srikanth, Toueg

4.4.1 Overview

Achieves clock synch in Byzantine setting.
Also, protocols to reintegrate a failed process (or, they say, integrate a new process), and to initialize the main algorithm.
The reintegration protocol is like that of LL, but the initialization protocol is different (seems more

elegant).

Their work builds on ideas in LM and LL.
But they don't do averaging and don't use Byzantine agreement.
Instead, they describe an algorithm that assumes authentication, and use a general fault-tolerant broadcast "primitive" they have developed to emulate the authenticated algorithm in a Byzantine setting.

4.4.2 Authenticated algorithm

The algorithm using authentication has built-in parameters P and α.
A process expects to resynchronize for the k^{th} time at logical time kP on its local clock.
When that time is reached, it broadcasts a (signed) round k message, indicating it's ready to resynch.
When a process receives such a message from $f+1$ distinct processes, it knows that at least one nonfaulty process is ready to resynch.
It is then said to "accept" the message, and decides to resynch, even if its personal logical clock hasn't yet reached kP.

Q: Shouldn't it send its own round k message in this case?

The process resynchronizes by resetting its logical clock to $kP+\alpha$, where α is the (small) constant parameter mentioned above.

No averaging! The whole issue here is to agree on *when* to set the local logical clock to a predetermined valued, not *what* value to set it to.

The process also relays all the received round k messages to other processes, to ensure that they also resynch very soon.

Because at each round, some nonfaulty process' clock had to reach the desired logical time in order for resynchronization to occur, we know that resynch only happens when it "should".
The relaying ensures that nonfaulty processes resynch within a short (real) time of one another.

Initialization is smoother here than in LL.
Namely, any process decides, independently, that it's time to start the clock up, and broadcasts a round 0 message.
When $f+1$ round 0 messages are received, a process "accepts" the message and does the first synch, setting the clock to α.
Then it relays all the round 0 messages that it has.

Then it just continues with the main algorithm for subsequent rounds.

4.4.3 Byzantine algorithm

For this, they use a broadcast primitive that they developed for the purpose of emulating authenticated Byzantine algorithms in a general Byzantine setting.

Describe the primitive as a service:
Invoke it with $bcast(m)$; messages are delivered with $accept(m)$.
The primitive is supposed to guarantee:

1. If at least $f+1$ nonfaulty processes bcast m, then every nonfaulty process accepts m within a short time.
2. If a correct process accepts m, then every other correct process accepts m within a short time.
3. If no nonfaulty process ever bcasts m then no nonfaulty process ever accepts the message.

This can be implemented in a Byzantine setting, with the usual $3f+1$ processes, using an echo algorithm (you saw something like this in the Malkhi-Reiter paper).

Specifically, when a client submits $bcast(m)$, the implementing algorithm sends around an $(init, m)$ message to everyone.
Any implementation process that receives $(init, m)$ from at least $f+1$ distinct processes sends $(echo, m)$ to everyone.
Any process that receives $(echo, m)$ from at least $f+1$ processes resends $(echo, m)$ to everyone.
Any process that receives $(echo, m)$ from at least $2f+1$ distinct processes issues $accept(m)$.

This implementation is shown in previous ST paper to guarantee the three conditions above.
Seems clear.

Can use this to convert the simple clock synch algorithm described above so that it works for general Byzantine faults:
Replace the signed communication with use of the broadcast primitive.
We no longer have to relay messages as in the authenticated algorithm, since that is now handled within the bcast primitive.

The reintegration and initialization protocols also work with this Byzantine extension.

4.4.4 Discussion

Nice, elegant, modular.

They also prove that their algorithms achieve the "optimal" accuracy (closeness to real time) that is possible given the drift rate and other parameters.

Q: They claim that their reintegration protocol can be used to allow a new process to join the system. That would be a nice "dynamic" property of the sort we're trying to study in this course.
How would this work?
The algorithms seem to depend on knowledge of who the participants are, e.g., in looking for messages from $f+1$ out of some larger set of possible senders.
Should we look for $f+1$ out of the whole universe?
With an arbitrary set of possible senders, the f limitation might not be reasonable.
What exactly happens during a joining protocol.

It does seem plausible that this could be made to handle arbitrary arrivals—more easily than the algorithms that depend on taking an average. An average depends on knowing n, whereas looking for the arrival of a particular logical time at some nonfaulty node need not.

4.5 Dolev, Halpern, Simons, Strong

Dynamic Fault-Tolerant Clock Synchronization.

This paper assumes Byzantine failures with authentication throughout. The algorithm seems something like Srikanth and Toueg's main algorithm.

An interesting wrinkle here is their focus on dynamic aspects of the problem, specifically, letting entirely new processes join the clock synch algorithm midway through.

The description of the joining parts of the protocol seem somewhat complicated to me—low-level and detailed. I won't try to present the details.

At a very high level: They have the processes keep track of the "members" of the system in state variables. These state variables are replicated everywhere.
The nonfaulty processes need to agree on who the members are at each logical time.
They do this with a separate (?) fault-tolerant replicated data service.
This service is implemented on top of the logical clocks—using essentially a synchronous model.
Thus, clock synch, joining, replication, are all interrelated in somewhat difficult ways.
Not easy—probably there is more to do to make this look simpler.

They need various assumptions to make sure that the system maintains appropriate connectivity in the face of joining and failures.

Looks interesting, but LTTR.

5 Lower bound on number of processes, with Byzantine failures

Several of the algorithms—by Lamport and Melliar-Smith, Lundelius and Lynch, Srikanth and Toueg—tolerate Byzantine failures and incur a corresponding $3f + 1$ cost in number of processes.
This should look familiar: $3f + 1$ processes are needed to tolerate f failures when solving various other consensus-type problems in the Byzantine setting, including ordinary Byzantine agreement and weak Byzantine agreement.
The same number arises here, as a lower bound.

This lower bound was proved first by Dolev, Halpern, and Strong.
But the proof in my "Easy Impossibility Proofs" paper with Fischer and Merritt is better: simpler, and gives more insight.
It uses the "big ring" idea that you might recall from the weak Byzantine agreement lower bound result.

Assume n processes in fully connected reliable network, message delivery time in range $[0, d]$.

f processes may exhibit Byzantine faults.

Processes have hardware clocks HC_i, which stay within a linear envelope of real time, specifically:
At any time t, and for any i, HC_i is in the range $[c_1t, c_2t]$.
Thus, we are assuming that all the HCs start at 0 and may drift apart only later.

Assume that local processing is scheduled deterministically according to the HCs.

We require that the processes produce adjusted clocks AC_i, as derived variables of their states. These should satisfy:

Agreement: There exists a constant ϵ such that, in all reachable states, for all nonfaulty processes i and j, $|AC_i - AC_j| \leq \epsilon$.

Validity (linear envelope):
There exist constants a_1, a_2, b_1, b_2 such that, at any time t, and for all nonfaulty i:
AC_i is in the range $[a_1t + b_1, a_2t + b_2]$.

Theorem: There is no algorithm in this Byzantine model, with $3f$ processes, that achieves Agreement and Validity and tolerates up to f Byzantine failures.

Proof: Do for 3 vs. 1; the ideas are the same for $3f$ vs. f.
Assume such an algorithm exists, and the processes are numbered 1, 2 and 3.
Choose a number k that is "sufficiently large" with respect to $a_1, a_2, b_1, b_2, c_1, c_2, \epsilon$.
We'll see how big later.

Construct a ring S of processes $1, 2, 3, 1, 2, 3, 1, 2, 3, ...1$ with $2k$ processes.

To specify the behavior of S, must describe the HCs and the message delays.
(Also need to resolve race conditions, need some default.)

Start at some process, say, indexed by 0, let HC go at the slow rate c_1, that is, its value at time t is c_1t.
And let its outgoing messages take time exactly d (the max).

For successive nodes further and further away in both directions, use successive speedups by $C = c_2/c_1$.
That is, the immediate neighbors have HCs that go at rate $c_1C = c_2$, that is, their values at time t are c_2t, and their outgoing messages all take time exactly d/C.
The next neighbors have HCs going at rate c_1C^2 and message delays d/C^2.
And so on.

At the opposite pole in the ring, indexed by k, we get HCs going at rate c_1C^k and message delays d/C^k.
That HC may look funny, because it's too fast. But note that the processes involved can't tell that it's too fast—all they see is their HC values, not the relationship between the HC values and real time.

This ring S executes somehow, though there are no special correctness requirements in this execution, since it doesn't satisfy the HC bounds.

Now consider any pair of consecutive nodes in the ring.
It looks to them as though they are nonfaulty and in a "triangle" system in which the third process is very faulty.
That process is imitating the entire rest of the ring.
Though their clocks are going very fast, they can't tell this.
They just "think" their clocks are going at a normal rate, with their messages taking a (normal) long time.
But actually, their clocks are going really fast, and their messages are incurring very short delays.

Because they think that they are in a triangle acting correctly, so by agreement, they always maintain their AC values within ϵ of each other's.

Now consider a very large real time t.

Consider the slowest process, indexed by 0, and its clockwise neighbor, indexed by 1.
It looks to them as though they are in a triangle system, with their hardware clocks going at the same rate as specified.
(The slowest couple of processes have HCs that are progressing at allowable rates.)
So by validity in that ring, at the chosen time t, $AC_0 \in [a_1 t + b_1, a_2 t + b_2]$.

Now consider the fastest process in the ring, process k, and its clockwise neighbor, process $k+1$.
Process k's HC goes at rate $c_1 C^k$, and its neighbor $k+1$'s HC goes at rate $c_1 C^{k-1}$.
It looks to them as if they are in a triangle system, but where the computation is "scaled" (slowed down) by a factor of C^{k-1}.
It's real time t in ring S, but in this scaled computation, the "real time" is much bigger, namely, $t' = tC^{k-1}$.

Now we can apply the validity property to the scaled triangle computation, to conclude that AC_k at time t' is in the range $[a_1 t' + b_1, a_2 t' + b_2]$.

So, speeding it up to what it is in the S computation, we see that AC_k at time t is in the same range $[a_1 t' + b_1, a_2 t' + b_2]$.
Now plug in $t' = tC^{k-1}$.
That is, in S, AC_k at time t is in the range $[a_1 tC^{k-1} + b_1, a_2 tC^{k-1} + b_2]$.

Now we can get a contradiction by putting together the various constraints that we've shown about S behavior.

We see that, at time t, $AC_0 \leq a_2 t + b_2$.
Also, at time t, $AC_k \geq a_1 tC^{k-1} + b_1$.
But by repeated use of the agreement bound for neighbors, we can deduce that $AC_k \leq AC_0 + k\epsilon$.
This says that $a_1 tC^{k-1} + b_1 \leq a_2 t + b_2 + k\epsilon$.
For fixed $a_1, C, b_1, a_2, b_2, \epsilon$, this is violated by choosing sufficiently large t and k.