Implementing a Framework for Certified Grid Computing

Margaret DeLap

May 3, 2002

Abstract

Large quantities of computing resources go unused every day; many PCs, for example, are idle much of the time. Grid computing aims to tap these resources in much the same way as we use an electric power grid. This would allow us to solve large, computationally expensive problems cheaply. However, on large networks such as the Internet, it is not possible for all users to know and trust one another. Further, since resource donors may not receive any compensation, they have little incentive to accept code without confirmation that it will be safe to run. For these reasons, the ConCert project seeks to provide a theoretically sound basis for grid computing in an untrusted environment. To show that such a goal can be feasible, this project provides an initial implementation, using x86 TAL (Typed Assembly Language), of an actual framework for acquiring, verifying, and running grid code. TAL provides the ability to verify type safety; later, ConCert hopes to deliver capability to prove other properties, such as bounds on resources used. For this initial framework, requirements include taking full advantage of the fact that it will not be necessary to trust the code provider, tolerating failure in the face of unreliable links, and distributing code among machines differing in computing power.

Contents

1 Overview

The proliferation of computers, especially personal computers, has meant that vast quantities of computing resources go idle much of the time. It has long been thought that these idle machines might prove to be a cost-effective source of cycles if there were an effective way of tapping them. Indeed, there are several projects that have already done so. Examples include the use of computers on the Internet to simulate protein folding[Fol00] and climate change[Cas99], to assist in AIDS[Fig00] and cancer[CAC00] research, and to find prime numbers[GIM96] and aliens[SET01]. Interest in this type of *grid computing* application appears to be increasing, but at the same time there are factors that discourage users from participating. In the presence of viruses, worms, Trojan horses, and other malicious programs, running code from unknown sources is understandably unappealing.

For the purposes of this document, we will define grid computing as the use of large networks of computers, on a volunteer basis, to solve computationally intensive problems. A user who wishes to donate cycles to help solve these problems must acquire the developers' code and run it for them. Under the prevailing circumstances, such users usually download the necessary code from a server on the Internet, implicitly trusting:

1. that the code provider has good intentions;
2. that the code provider has succeeded in carrying them out, by writing sufficiently correct code;
3. that the code has not been replaced, on an unsecured server, with something malicious.

All of these assumptions must apply even to code that people use for their own benefit. If knowledgeable users are wary of running unknown code from arbitrary sources even for their *own* utility, then it is all the more important to be able to show users that grid code is safe, since they usually do not get any direct benefit from running it.

In this document, we discuss what certified code might have to offer for purposes of proving that code is safe; advantages of using certified code in the grid setting; and what implications certification might have for grid programming. We detail an implementation of a framework for grid computing that attempts to take into account some of these implications, and characterize what applications might be suitable. Finally, we discuss some of the further work to be done in making this framework more robust and powerful.

2 Security policies

At this point we describe what sort of assurances a user might want from others' code, and suggest what might be involved in specifying them.

One assurance a user might want would be that a program would not exceed some bound on the use of a resource. Such resources might include CPU time, memory use, disk use, peak or total network bandwidth use, number of child processes created, etc.

Another issue is that of protection. Ideally, we would like to see certification of properties of a finer grain than that generally available through traditional "user" or "group" permissions. A user may own a file without wanting code he or she runs to be able to read it. Similarly, users may have access to printers without wanting to allow miscellaneous, unauthorized code to print documents on them.

Third, the user may want to ensure that the program will not permanently alter the state of his or her machine. For example, if a program creates temporary files, the user probably would want them deleted before it exits so that they would not clutter the disk.

3 Certified code

3.1 Definition

Certified code comes with some form of proof that it adheres to certain properties. Examples are proof-carrying code[NL97] and typed assembly language (TAL)[Mor99]. TAL, which is what this project uses at this point, provides type safety. A long-term goal of the ConCert Project[Con01] is to investigate what other properties can be certified. Bounds on resources such as CPU time or filesystem use would be of interest with respect to users' security policies. It is not yet clear to what extent these can be certified statically, as opposed to through the use of runtime checks. Some other projects use a significantly different approach. The goal of CoVirt[Cov01], for example, is to use virtual machines to run and examine the behavior of code without endangering the host operating environment. If the code turns out to be malicious, it (and the virtual machine as well, if necessary) can easily be terminated. Rather than using a virtual machine, TAL allows us to run the code natively after it has been verified; the disadvantage is that as of yet TAL runs only on Intel architectures.

3.2 Use on the grid

With certification, it is possible to verify that code has certain properties — for the moment, type safety. To the extent that code can be certified, it is no longer the case that the user needs to blindly trust the code provider.

The points of trust noted above are all, in some sense, related to the programmers and system administrators involved in the given code and its dissemination. The current situation practically prohibits users from running code from any other than the well-known, trusted projects. To say nothing of the advantages developers would receive from the increased information about their own code that certification might be able to provide, proving properties of grid code may well encourage more even sharing of resources. To the extent that we can examine and verify the code itself, the trustworthiness of the people involved in providing it becomes irrelevant. This allows developers more widespread and equitable ability to get others to run their code; they no longer have to be on well-known projects like SETI@Home to convince people that running their code is safe.

In addition, this means there does not need to be some central server providing code. Everyone who wants to participate can serve code to others independently. This should increase performance and reliability since we can now remove bottlenecks and single points of failure.

4 Concomitant challenges

Of course, there are complications involved both in this decentralized arrangement and in grid computing itself. Possible solutions to these complications will be discussed later in this document. For the moment, here are some things to consider:

4.1 Decentralization

In a centralized architecture, volunteers presumably know *a priori* where to look for the code: the central server of the project they want to work on. In a decentralized architecture, it becomes necessary to find other participating hosts. If the network is stable and unchanging, this is easy. On a highly variable network such as the Internet, discovery of other participants must be done at runtime and is no longer trivial.

4.2 Network characteristics

Once a node finds peers, it cannot assume they will always be available. The links interconnecting networks of workstations are much slower and less reliable than those connecting, say, a multiprocessor machine such as a Cray or an Origin. In general, this means that we need to tolerate failure and to minimize communication. This, too, is a problem associated with grid computing itself, not just the certified kind.

4.3 Malice

On public networks, various types of malice may assert themselves. One is that a "volunteer" may pretend to run given code, but actually just present a fabricated result. Though the volunteers may be protected by certified code, developers are not. Another problem is the potential for developers to swamp the network with useless code, although the definition of a worthwhile investigation is rather subjective — for example, some might say searching for aliens is not useful.

5 Architecture and implementation

To take advantage of the decentralization opportunities that certification provides, we use a peer-to-peer architecture. One example of such an architecture is Gnutella[GPS]. Gnutella is oriented toward sharing files while the ConCert framework's purpose is to share cycles. Nonetheless, both are intended for use on large networks including the Internet, so Gnutella's protocol has some ideas that are of use to us. To deal with the grid's unreliability, we also use some ideas from CilkNOW[BL97].

5.1 Parallelism model: cords

Our parallelism model bears a strong resemblance both to the CilkNOW model[BL97] and to dataflow computer architectures[CSG99], in which programs are represented as graphs of dependencies (Figure 1). Every node in the graph represents a function; it has one output, and some number of inputs. As soon as all inputs are available to a function node, it can run and produce the data for its output edge.

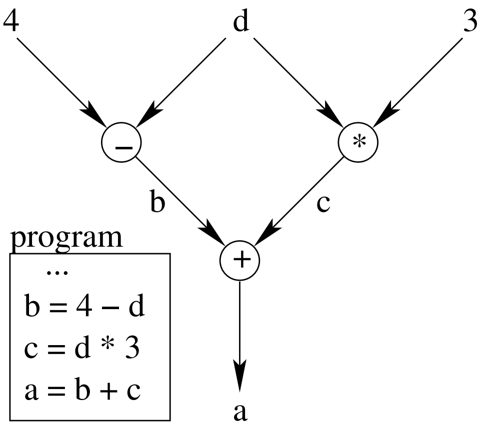

Figure 1: Example of a dataflow graph. Once `d` is bound, the `-` and `*` nodes can proceed, providing the values for `b` and `c`. At that point, the `+` node can run, giving the end result, the value of `a`.

This is an appropriate model for several reasons. First, it avoids use of shared memory, which would likely be disastrous for performance on a network with slow links, to say nothing of the difficulty of implementing it safely and correctly. Second, it discourages the use of fine-grained, high-communication threads, which would also be likely to exhibit poor performance on slow, unreliable networks. Third, it provides for simple scheduling, an important consideration where every participating node contains a scheduler. Finally, it simplifies failure tolerance, which will be discussed below.

Programs are represented as a graph of data dependencies. The unit of work in the system is called a "cord" (as it is something like a thread or process). Cords can be shipped from one node to another, but, barring failure, each cord is run to completion on only one host; it is not split up, nor migrated while it is running.

Cords may depend on results of other cords. In that case, the needed results are given as the arguments to the cord depending on them. Cords can have both *and* and *or* dependencies. From each set of *or* dependencies, the cord needs only one result before it can run. From each set of *and* dependencies, it needs the results for all of the dependencies in the set.

Figure 2 shows an example of a dependency tree. Cord `foo` depends on both of the two cords on the left, but only one or the other of the two on the right. Since the leftmost required cord is done and its result is available, the argument specifying that result, namely 4, has been filled in in `foo`.

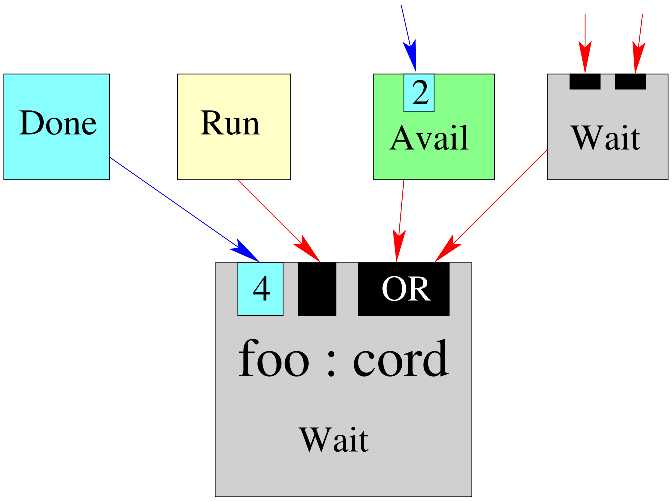

Figure 2: A sample cord dependency tree. Unfilled dependencies are black; filled ones contain hypothetical result values 4 and 2.

5.2 Components

In the underlying system, all participating nodes are symmetric. In other words, each is able to both serve work to other machines and to do work for others.

The term *conductor* refers to a node in its role of serving cords; it does not mean that said node cannot be running someone else's cord at the same time. Each node on the system has three main components or roles: *conductor*, *worker*, and *locator*. Figure 3 provides an overview of the node and its components; we will go on to discuss the components in detail.

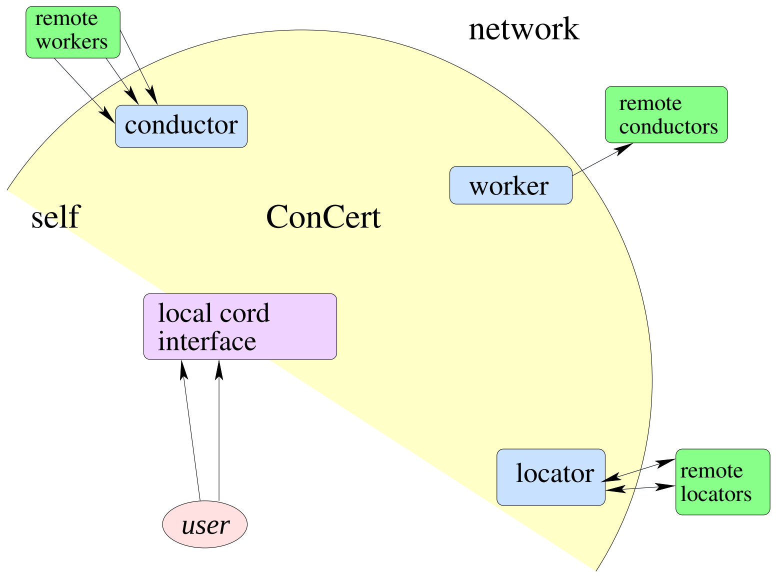

Figure 3: ConCert components

5.3 Discovery: locator

The central idea of Gnutella, which we apply in our implementation, is that a node should be aware of others within a certain “horizon.” To accomplish this, a node (“A”) as locator sends out a ping message to a few nodes it knows about initially; if any of these are up, they respond and also forward A’s ping to any machines they know are alive. To keep the network traffic within reasonable limits, the Gnutella protocol limits the distance a ping can travel to 5-7 hops. Here, “hop” refers to communication between nodes in the protocol, rather than a hop in the routing sense of the word. Whenever a node receives a ping or a response, it updates its list of known active nodes accordingly.

The `locator` component works in much the same way. Figure 4 is a slightly more detailed view of how the locator fits in to our software. It communicates with other locators on the network to discover IP addresses of more participating nodes; it then makes these addresses available to the rest of the ConCert system.

One special feature of our discovery protocol is that it incorporates some support for allowing nodes to use legacy versions of other protocols, most im-

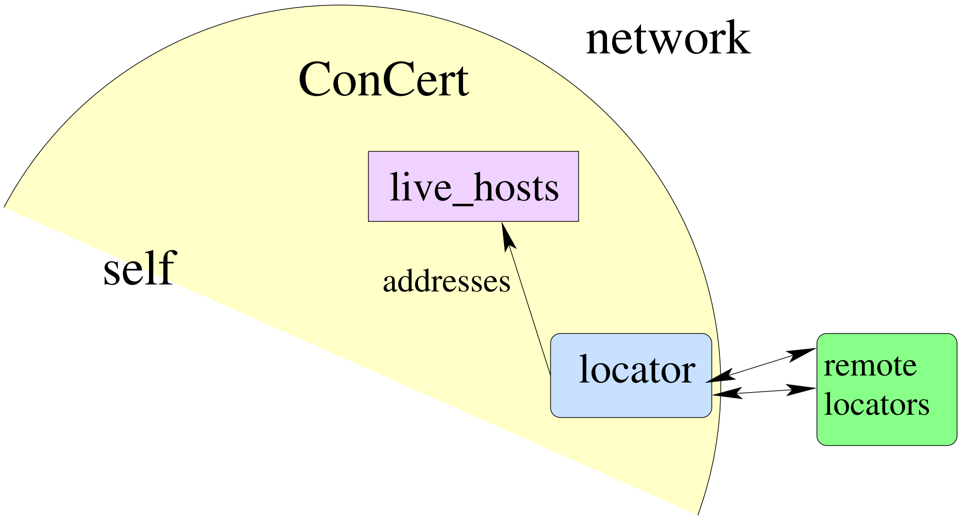

Figure 4: The locator component. “self” indicates the entire host on which it is running.

portantly the conductor-worker protocol (Sections 5.4 and A.2). As well as finding peers, during discovery nodes also negotiate which protocol versions they will later use to exchange work.

As in Gnutella, locators must be primed with the IP address of at least one other participant on the network; this information may be saved in a file.

5.3.1 Protocol example

Figure 5 shows one possible interaction between locators. In this example, `fred` has just connected to the network. The other nodes are already aware of each other. Initially, `fred` knows only about `xena` — `fred`’s host file contains `xena`’s contact information.

The conversation could proceed as follows:

1. `fred` sends a `HELLO` to `xena`. The immediate effect is that `xena` adds `fred`’s information to her list of live nodes that speak a compatible version.

2. `xena`, who speaks protocol version 2, proceeds to respond with a

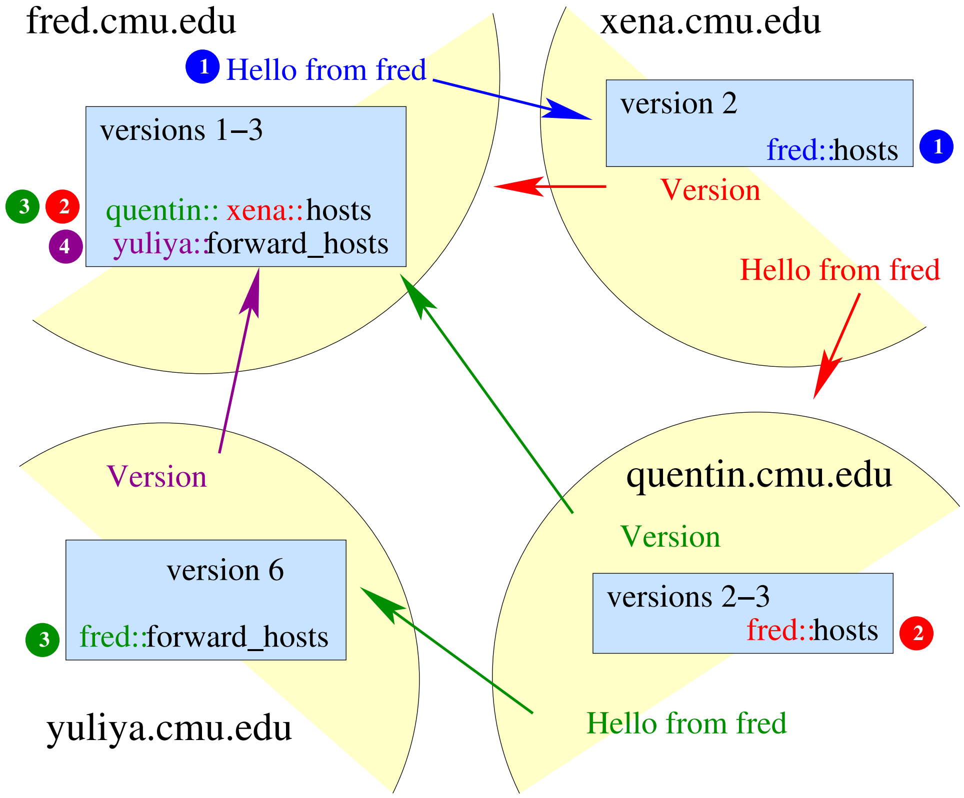

Figure 5: A sample conversation between locators

VERSION to fred and forward fred's HELLO to all of her known live nodes, aside from fred. The effects are:

- fred adds xena's information to his host list.
- quentin, who receives the forwarded HELLO, sees that fred is compatible and adds fred to his list.

3. quentin sends his preferred version directly to fred and also forwards fred's HELLO to yuliya. Then:

   - fred adds quentin to his list, and
   - yuliya, who does not speak any compatible versions, adds fred to her list of forward_hosts — she can at least forward HELLOs to fred anyway.

4. yuliya sends a VERSION message to fred. This will not match any of fred's versions, but enables fred to forward HELLOs to yuliya. fred adds yuliya to his list of forward_hosts.

This sequence brings up an interesting point: wouldn't it be the case that both quentin and xena would forward fred's HELLO to yuliya? This is, in fact, what happens; in the example, quentin's forwarded message just happens to arrive first. Currently, yuliya's strategy on the second receipt of the HELLO would be to just send it along anyway, but not send another version message to fred. Whether the possible increase of connectivity is worth the higher traffic is open to question. Given similar network latencies between nodes (a dangerous assumption for Internet grid computing), earlier messages would be likely to have more hops left to go than later messages would. This would mean that if we were to drop later messages, they wouldn't have gone as far anyway. Since there is great variation in link latencies on the Internet and since these messages are quite short, it seems worthwhile to send them along even if they are duplicates.

5.3.2 Anonymity versus traffic

One major difference between this protocol and Gnutella is that the VERSION responses are sent directly back to the originator of the corresponding HELLO. In Gnutella, responses are forwarded back through all intermediate machines; ping messages contain a randomly generated identifier rather than an explicit IP address, so lookups have to be done at each hop to find the previous sender on the path.

Gnutella's purpose in doing this is to increase anonymity at the cost of increased traffic. The rationale seems to be that one might want to search for files without letting people connect the search terms with the IP address they came from. Once people decide to download files, though, they connect directly to the source hosts and thereby reveal their IP addresses.

This architecture does not have a search protocol, so there is no need to preserve anonymity for searches. There do exist people who collect IP addresses for the purpose of scanning large numbers of hosts for vulnerabilities. However, since peers in this architecture spend relatively large amounts of time connecting directly to each other to request work, it seems unlikely that increasing `HELLO` anonymity will make much of a difference in preventing this.

5.4 Propagation of cords

Now that nodes are in communication with one another, cords need to be sent around the network. First, cords are "injected" into the system locally using the protocol specified in A.3. Submitted cord specifications go to the conductor on the machine where the submitting was done, either by a user or by already-running cords that need to create another cord to proceed. From the specification, the conductor creates a cord (Figure 6) containing "pointers" to the necessary TAL files — the code itself, as an object file, and the typing annotations used to verify the code.

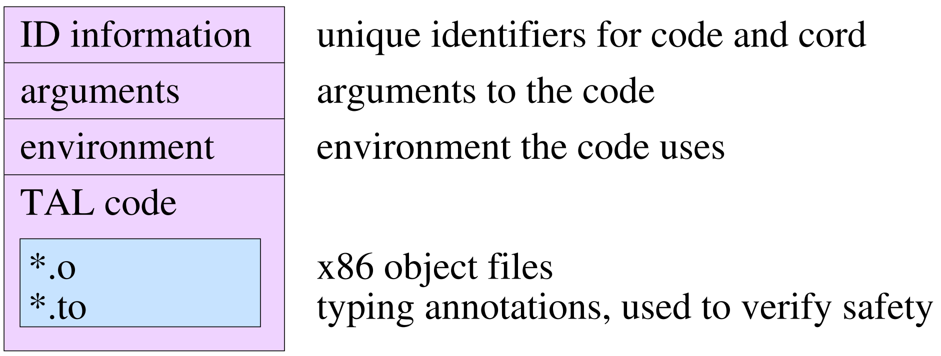

Figure 6: Cord internals

Conductors are passive, so the above is all they do until they get a request for a work. In other words, cords are always pulled from the network. This allocation method is closely related to the parallel programming concepts of *dynamic assignment*[CSG99] or *work-stealing*. Due to the changeable nature of the grid, any static form of scheduling is impracticable. In addition, having workers initiate work transfers has two notable advantages:

1. It reduces unsolicited traffic. Workers will probably want to run at most only a few programs at a time; having conductors bombard them with requests to do work would be at best ineffective and at worst encourage them to leave the network. On the other hand, conductors would more often be happy to have several workers connect to them at once.

2. It provides conductors the opportunity to try to fit the cords they send out to the security policies of the conductors that request them. This has the potential to reduce unnecessary network traffic.

Precise details of the protocol[Dun01a] used to accomplish work exchange are presented in Section A.2; for now we will describe the essential workings.

Nodes may request work from any live host they know of. Figure 7 is an example of how this may be done. The worker, running on a host named `lackey`, contacts the conductor on `boss` directly, sending the parameters of its security policy. `boss` will try to find a cord matching the policy. If it does, it sends back information regarding that cord: a unique identifier for the cord and for the code, and the arguments to be used. If `lackey` determines it does not already have the code on its filesystem, then it communicates with `boss` again to request and receive the code. The conductor looks up the applicable TAL files, archives them into a format that can be shipped over the network, and sends them. Once that is done, the worker can verify and run the cord, sending back either the result or notification that the cord was not completed. For example, the user might decide to reclaim the machine and kill off running cords.

A consideration the conductor-worker protocol does not address is the size of datasets. We avoid sending code over the network where possible, but we do not do the same with datasets. For some problems, datasets could conceivably be very large; in that case, it might be advantageous to cache them on the worker's machine so that if they were ever used again, it would not be necessary to reacquire them. On the other hand, users might wish not to cache such large files on their disks if they will not necessarily be used. While this applies also to programs, it seems more likely that the classes of problems suited to this architecture (detailed in Section 6) will

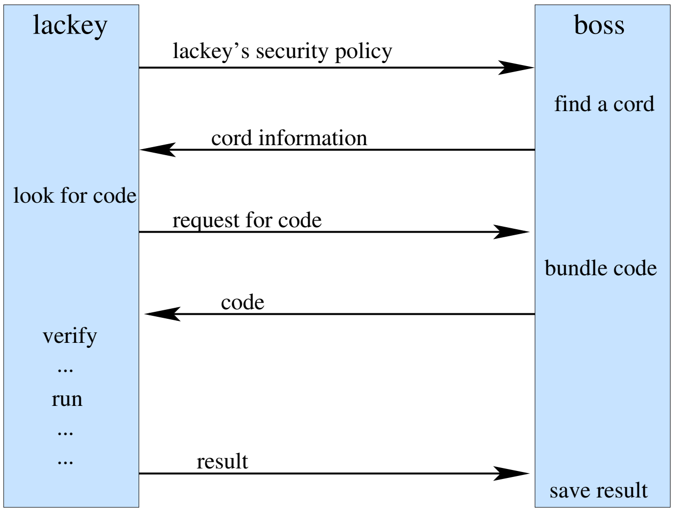

Figure 7: Example conversation between a worker and a conductor (based on [Dun01b]).

run the same code repeatedly on varying data rather than varying code on one dataset.

5.5 Scheduling

5.5.1 Conductor

Our architecture requires that only cords without unfilled dependencies can be sent out to run. Therefore, a conductor maintains separate queues for ready and waiting cords — waiting cords are those with dependencies that have not yet been satisfied. These queues are updated whenever a new cord is submitted or a result is returned. When a cord is submitted by a user, its component cords cannot have any dependencies to start with, so they go directly to the ready queue. Cords submitted by parent cords, on the other hand, may have unresolved dependencies and in that case will go into the wait queue. A returned result may induce the conductor to move a cord from the wait queue to the ready queue if it is the last unfulfilled dependency that cord has. Finally, it may be useful to have results themselves linger on the conductor so that users may query for them. Other conductors may also choose to cache the results of queries in order to make the network more failure tolerant. Failure will be discussed further in section 5.6.

Figure 8 gives a scheduling example. Here, a running cord finishes; the conductor saves its result, and updates the arguments of a waiting cord that depends on it. Since that cord has no other unsatisfied dependencies, it becomes available to run.

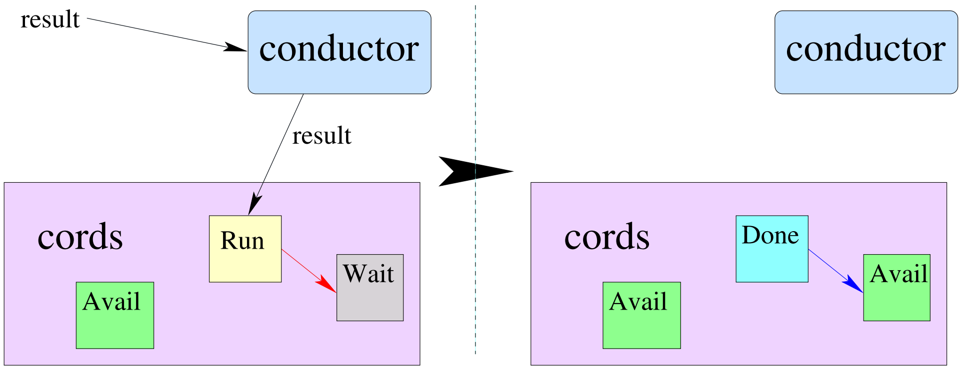

Figure 8: Conductor scheduling example

5.5.2 Worker

The worker must decide in what order to try to connect to known conductors. It can gather some information about conductors in the process of running: whether or not a conductor had cords available during previous connections, and if so, whether the cords could be verified as matching the worker's policy. Generally, it would seem that the worker should place more priority on conductors known to have cords previously, and perhaps not attempt to connect at all to those conductors that served non-verifiable cords — the desirability of doing this last is further discussed in Section 8.5.

To accomplish this, we maintain two queues: an active queue and an inactive queue. The worker runs indefinitely; on each loop it tries to make a connection and run a cord. First, it checks the active queue and tries to connect to the the host at the head of the list. If the worker fails to connect or the server has no cords, then that host is moved to the inactive queue. Otherwise, it is moved to the back of the queue to maintain a round-robin order on the active queue. Additionally, if the active queue is nonempty, the first host on the inactive queue is promoted to the end of the active queue so that connection can be retried. This round-robin strategy is intended to help distribute unequal computing resources more evenly, so that one program does not always get its code run on a slow or a fast computer.

If the active queue is empty, the worker next examines the inactive queue. If it fails to get a cord from the first host there, it will wait with a certain backoff before trying again on the next. The same thing is done if the inactive queue is empty. The backoff starts at 1 and is doubled, within limits, every time either of these occurs. It is intended to prevent possibly excessive connection attempts that could cause conductor hosts to be overloaded.

Alternatively, if the worker succeeds in running a cord from an "inactive" host, that host is moved to the active queue and the backoff is reset. In any case, if the inactive queue is nonempty, the first item is moved to the back after the connection attempt, same as in the active queue.

5.6 Failure

There are many possibilities for problems: network unreliability, instability of individual machines, malice, etc. One way to make failure recovery easier is to make all cords restartable. To do this easily, we impose some invariants:

- Any possible run of a cord is "as good as" any other. This is obviously true if the cord is deterministic; if it is not, some care must be taken.
- Cords do not communicate with each other directly; all communication is done through dependencies.

These obviously affect how the programmer is going to think about writing code for this framework. Tom Murphy and Evan Chang are working on a higher-level abstraction that will partition thread-like "tasks" into cords.

In any case, these invariants allow cords to be restarted if necessary. Deciding what is meant by "necessary" can be difficult. If a node running a conductor's code decides to stop doing so and helpfully notifies the conductor in question, then that conductor can easily reschedule it. If, however, the conductor is not notified, then it must determine when to assume that something catastrophic has happened and reschedule the cord. The solution seems to be a system of keepalives sent between the cord and the conductor, but preferably we want to hide this from the programmer; it would be tedious for a human to write and hence error-prone.

Malice may result in fabricated results being sent back to the conductor. Since ways of checking results can differ vastly between applications, result checking should be left to the application developer. The way SETI@Home deals with this is to (1) keep its code obscure and (2) serve the same code to a few different hosts to decrease the likelihood that all of them will give a fake answer. That is one solution; another is to run a randomly-selected subset of one's own cords and check the answers against those given by other hosts. Of course, this works only for deterministic cords.

6 Suitable applications

Many of the desiderata for an application for this framework are the same as those for any parallel program. Ideally, we want to minimize intercommunication — in our case, dependencies; balance the workload; and keep overhead as low as possible.

There are performance and reliability differences between the grid and a normal (e.g. bus-based) multiprocessor computer, however. The most obvious are the increased latency and unreliability of the network. This means it is all the more important to reduce dependencies. At the same time, we do not want to go too far — the program with no dependencies is no longer parallel.

We also have to contend with the fact that the nodes of the grid vary in processor speed, memory capacity and latency, link bandwidth, and other characteristics that affect performance. This certainly affects load balancing, in some cases severely. As of yet, the framework contains no mechanism for dealing with this issue, but it may not be as pressing as one might think. In scientific computing, for example, developers are usually more interested in the feasible problem size than in the amount of time the program takes to run (within reason).

6.1 Common models for parallel computation

We will now consider a few common parallel programming models and how effectively programs written using them might translate to our cord-based model.

6.1.1 Shared memory

One very common model is known as *shared memory* or *shared address space* (SAS), in which processors communicate by sharing and writing to the same space. Unfortunately, this represents a very different way of thinking about parallelism that does not map well to our model. Emulating shared memory is a difficult enough problem on distributed-memory machines designed to do it; doing so on a network of workstations would, with current technology, be ruinous for performance. Such a concurrency model might also make it difficult to prove properties of the individual threads involved.

6.1.2 Message passing

Fortunately for us, much scientific computing is instead done using the *message passing* model, in which communication is done by sending explicit messages. For example, essentially all work done on the Pittsburgh Supercomputing Center's Terascale Computing System uses message passing[Sco02].

Communication on the grid maps more naturally to explicit message passing than it does to the SAS model. To translate these messages to inter-cord dependencies, we must partition threads at their `receive` and `send` points; the messages the threads are expecting and sending become dependencies and results, respectively. As shown in Figure 9, we must also assume that the segments of the threads between sends or receives are sequentially dependent.

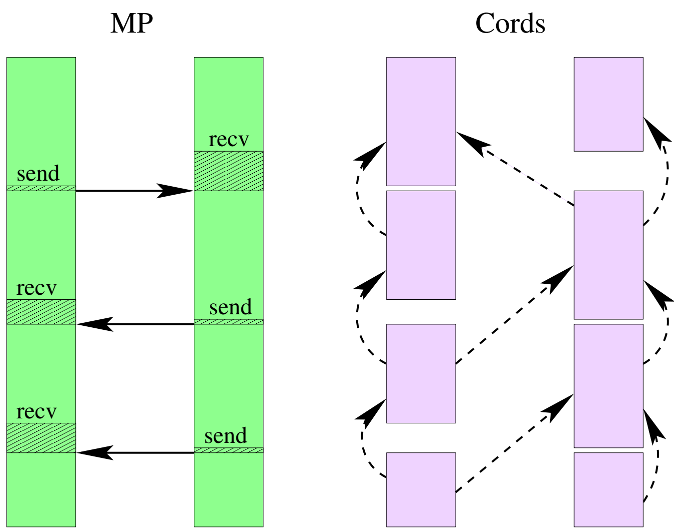

Figure 9: Translating message passing threads into cords. The messages between the threads on the left become dependency arrows on the right.

Splitting threads into cords will, in some cases, give us somewhat more flexibility in scheduling. One practice we must avoid for the sake of performance is extremely *fine-grained* message passing involving large amounts of communication. While the grid's network latency is discouraging enough, we also bring in the overhead of encoding, scheduling, and transferring cords. We should create enough cords to take advantage of parallelism, but not so many that this overhead overwhelms us.

6.1.3 Data parallel processing

The *data parallel* model was characterized by Flynn in [Fly72] as *single-instruction-multiple-data* (SIMD) — one instruction applied by multiple processors to multiple data. Thirty years later, we can think of an "instruction" as instead a "function" or "program" and distribute the data to multiple workstations. Such a model could be ideal for our architecture for a few reasons.

First, each node need acquire the program only once. This saves time spent marshalling, sending, and unmarshalling code.

In our (certified) case, it can also save time spent verifying code; if a program has been verified once and assuming the user's filesystem is secure — something we must assume in trusting the verifier in the first place — it need not be verified again.

Translating a data parallel program into cords may seem somewhat more involved than translating a message passing program. This is because the function that is applied in parallel can involve a communication step, such as shifting data from one processor to the next. However, the same procedure used in the case of message passing applies here — turn the communication step into a result and a dependency.

It turns out, then, that for our purposes data parallel is effectively a nice special case of message passing. This means message passing is a more general and therefore sometimes more useful model, but wherever possible reusing code can be good for performance. As in message passing, we still need to take care not to use too many communication steps.

6.2 Current work

At present, a few applications have been written for this framework, including a ray tracer by Jason Liszka and other graphics programs by Tom Murphy. At a higher level, Evan Chang has written a parallel theorem prover for linear logic[Cha02] that, it is hoped, will be mapped onto this architecture in the future.

6.3 General characterization and summary

Programs that will run effectively on our grid tend to have the following characteristics:

- a low ratio of communication to work. In other words, they are *CPU-bound* rather than *I/O-bound*.
- a shallow dependency graph. This essentially translates to having high levels of parallelism rather than many cords that must run sequentially.
- verifiable results, or identical, deterministic cords that are scheduled on multiple nodes. This will counteract the possibility of fabricated results.

7 Conclusion

Certified code may increase developers' opportunities to use grids such as the Internet through reducing the need for trust. Steps involved in making this possible include designing a suitable architecture, developing capability to verify properties of programs, and providing tools for programmers to use the certified code framework effectively. This work represents one architecture design that makes use of unique advantages certified code gives us, while providing for a programming model with some level of natural support for failure tolerance.

8 Future work

This section describes features that do not exist in the current architecture, but which could conceivably be useful.

8.1 Repair vs. re-run

Under certain situations, it may be desirable not to try to re-run a failed cord. Instead, the application may want to arrange to approximate its result. For that purpose, failure handlers of some sort would be useful — in other words, if a cord fails, then its failure handler should be scheduled instead of rescheduling the original cord. In fact, the handler could be as simple as a default value for the result, rather than code that would need to be scheduled.

8.2 Checking results

We have addressed issues of trust in one direction: the resource donor does not need to trust the code he or she runs. Further investigation is needed with respect to the opposite direction, as it is possible for malicious users to fabricate results. For now, the developer must deal with this possibility. Here are a few possibilities for future work:

- Some results are self-certifying. For example, proofs generated by a theorem prover can be checked.
- Cords can be run multiple times on several nodes. Unless they collude, seeing whether the answers match is a realistic, if not infallible, strategy. As was previously mentioned, this is what SETI@Home does.
- Tom Murphy is investigating cryptographic methods that would make it expensive for a node, once it has fabricated results, to reestablish a trusted identity on the network. This would probably tie in with future work on authentication.

8.3 Scalability

Gnutella is known to have scalability problems[Cli00]. The major problem is that the lowest-bandwidth nodes become rate-limiting for approximately all connections on the network. This is likely due in large part to the fact that many of the connections on the Gnutella network are made indirectly, with several forwarding steps, to increase anonymity. In this ConCert implementation, all connections are direct except for those made by `HELLO`s. Further, we do not have anything like Gnutella's "search" capability. It therefore seems likely that we should be much less susceptible to this scalability problem. Further investigation is needed to confirm or refute this conjecture.

8.4 Authentication

Authentication would clearly be useful if developers want to reward cycle donors. Mutual authentication and single sign-on are already supplied by the Globus toolkit[Glo02] through the Grid Security Infrastructure (GSI).

Also, users probably want to avoid running "useless" cords. In other words, they will want to know that they are working on something they find reasonably valuable. For that purpose, authentication might be useful to control from which nodes or developers a worker acquires code. This, of course, means users have to trust the node or developer to supply only what they consider worthwhile code; proving to the general user that any given code is "about" a particular problem or will be used in solving certainly seems out of the question.

8.5 Blacklists

Another idea from Gnutella that might be useful for this architecture is that of blacklists. Possible uses include:

1. letting a worker avoid connecting to nodes from which it has received non-verifiable code
2. letting a conductor avoid sending code to a node from which it has received a fabricated answer
3. letting a locator avoid forwarding information to a node considered malicious, i.e. for one of the two reasons listed above

It would seem that to accomplish both items 1 and 3, all a host needs to do is drop the offending node from its list of known nodes. The problem with that is that if the offending node reconnects to the network, the host in question may, in fact probably *will*, get its `HELLO` again and reestablish relations with it.

We may want to avoid doing item 3 anyway. If we do not allow `HELLO` forwarding through the malicious node, we may be cooperating in denial of service. Item 2 could be especially useful to allow conductors not to bother with malicious cycle donors, but it is probably also necessary to have some way to be removed from blacklists. As an example, assume `evil.foo.org` and `good.foo.org` have dynamically assigned IP addresses. If `evil` gets on a conductor's bad side with some IP address, we still want `good` to be able to work for the conductor even if it later gets assigned to `evil`'s former address.

Additionally, we should consider the fact that code may fail to verify for any of several reasons. A programmer may have perfectly good intentions but need more resources for his or her program than a given worker is willing to donate. If that worker blacklists the developer's machine, it loses the opportunity to run future code that *could* comply with the worker's policies.

In conclusion, authentication is likely be able to accomplish everything we would want blacklists to do and more, without causing problems related to dynamic IP addresses. The likely reason Gnutella has used blacklists instead of authentication is that one of its aims is to preserve anonymity as much as possible; authentication, of course, has quite the opposite purpose.

8.6 Certification of other properties

Users are expected to be interested in more than type safety. Other assurances they might want were discussed in Section 2. Briefly, they included:

- bounds on resources: disk space, CPU time, network bandwidth, total network communication
- lack of access to use printers, read sensitive data, or delete selected files — such as any the program did not create
- cleanup of any temporary files the program did create

Work has been done on type systems for resource bound certification, notably [CW00], but further investigation is needed to determine whether and how certification of any or all of these properties is feasible in practice.

Regardless of the certification method, it will be necessary to encode users' security and privacy policies in such a way that certificates (or whatever is used) can be checked against them. For efficiency, conductors should be able to use encoded policies to determine which cords are worth sending to requesting workers: knowing a worker's policy, it would be senseless to send out a cord that the worker is not going to verify.

8.7 Timeouts and keepalives

Since the grid is full of potential for silent failure — a node may certainly reboot or otherwise fall off the network without notifying its peers — it is clear that the architecture needs some way of deciding when to give up on a given worker ever completing a cord. Absent certification of bounds on real runtime, runtime of a cord on a strange machine is unpredictable. For that reason, keepalives are our best bet. A reasonable implementation might be to have a conductor periodically poll all of its outstanding workers and wait for a response using a timeout.

For this purpose, the conductor would contact the worker process itself, not the process running the conductor's code. The worker would check for the existence of the process running the code and then report back to the conductor. This would mean the keepalive code would stay in the worker rather than burdening the application programmer. Although some controversy has surrounded keepalives with respect to whether they should be part of TCP, in this case it is clear that they should be part of the architectural protocols rather than of the applications. The cord model allows no communication; putting in keepalive communication would confuse matters severely.

8.8 Reduction of the trusted code base

In this implementation, the user is expected to trust the client/server software as well as the verification code. While it is not a very large program, the less code the user has to trust, the better. It would be ideal if, in a bootstrapping process, we could verify the client/server software as well. Whether this can be done along with more complex policies involving resource bound certification remains to be seen.

9 Acknowledgments

Frank Pfenning and Robert Harper, my advisors, provided a great deal of useful information and advice related both to the architecture itself and to this document. This work is done as part of the ConCert Project[Con01], in collaboration with Joshua Dunfield, Evan Chang, Tom Murphy, and Jason Liszka. Joshua Dunfield wrote the first version of the framework's code and protocols, much of which is still in use. Jason Liszka wrote a raytracer for use on the framework. Evan Chang and Tom Murphy developed a higher-level programming model for certified grid computing. Other members of the ConCert project supplied practical comments and suggestions.

References

[BL97] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, 1997.

[Cas99] Casino 21. `http://www.climateprediction.rl.ac.uk/`

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml.* Éditions O'Reilly, Paris, 2000.

[Cha02] Evan Chang. *Iktara in ConCert: Realizing a Certified Grid Computing Framework from a Programmer's Perspective.* Senior honors thesis, Carnegie Mellon University, 2002.

[Cli00] Clip2 Distributed Search Services. Bandwidth barriers to Gnutella network scalability. `http://www.clip2.com/dss_barrier.html`, September 8, 2000.

[GPS] Clip2 Distributed Search Services. The Gnutella protocol specification v0.4. `http://www.gnutella.co.uk/library/pdf/gnutella_protocol_0.4.pdf`

[CAC00] Compute Against Cancer. `http://www.parabon.com/cac.jsp`

[Con01] The ConCert Project. `http://www.cs.cmu.edu/~concert`

[Cov01] The CoVirt Project. `http://www.eecs.umich.edu/CoVirt`

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, January 2000.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach.* Morgan Kaufmann Publishers, San Francisco, 1999.

[Dun01a] Joshua Dunfield. ConCert implementation and protocols, first version. Created at Carnegie Mellon University, 2001.

[Dun01b] Joshua Dunfield. Presentation notes. Given at Carnegie Mellon University, September 17, 2001.

[Fig00] FightAIDS@Home. `http://www.fightaidsathome.org/`

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computing* C-21, pages 948–960, September, 1972.

[Fol00] Folding@home. `http://folding.stanford.edu/`

[GGF99] The Global Grid Forum. `http://www.gridforum.org`

[GIM96] The Great Internet Mersenne Prime Search. `http://www.mersenne.org`

[Glo02] The Globus project. http://www.globus.org

[Mor99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[NL97] George C. Necula and Peter Lee. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[JXT01] Project JXTA. http://www.jxta.org/

[Rep99] John H. Reppy. *Concurrent programming in ML.* Cambridge University Press, 1999.

[SET01] SETI@Home. http://setiathome.ssl.berkeley.edu

[Sco02] J. Ray Scott. Presentation at Carnegie Mellon University, April 25, 2002.

A Protocol details

A.1 Locator protocol

This protocol is used for communication between locators, the purpose of which is to discover peers. Version information is also included. At present there is only one version of the conductor-worker protocol (section A.2); in the case that that protocol is later extended or otherwise changed, this will allow both nodes speaking old versions and those speaking new ones to participate on the network. Although the semantics of discovering nodes have changed greatly, the format of the messages used is very much the same as in [Dun01a].

`HELLO(r1, r2, IP, TTL)` is sent out by every locator at startup. It is sent to other locators running on a predetermined set of hosts. Upon receiving this message, a locator forwards it to every other node it is aware of, and sends a `VERSION` message to the originating node — due to forwarding, this is not necessarily the node from which it received the message.

r1, r2 define the range of protocol versions, inclusive, that this node can speak.

IP is the IP address of the node originating the message.

TTL is the remaining number of hops this message is allowed to travel. It is initialized to 7 and decremented every time the message is forwarded.

VERSION(v) is the acknowledgment to a HELLO; it is sent to the address specified by the HELLO's IP field.

v is the number of the protocol version the sending host wishes to speak.

A.2 Conductor-worker protocol

The protocol used to transfer work and results is rather more complex than the discovery protocol described above. Since the worker actively pulls work from conductors, it initiates the conversation. This protocol is very much the same as that developed and used by Dunfield in [Dun01a].

A.2.1 Messages sent by the worker to the conductor

GETCORD(constraints) is the worker's request for a cord.

constraints is an encoding of the worker user's security policy. The conductor may use this to avoid dispensing work that the worker will reject as not compliant with the security policy. Sending out such work would waste time and network bandwidth.

GETPROGRAM(program_id) requests the code needed to run the computation. Since it is possible the worker already has the code, this message is not sent in all cases — programs are likely to be large and costly to send over the network.

program_id is a unique identifier that the conductor has computed from the code of the program in question. The worker knows this value as a result of receiving the conductor's CORD message (see below).

RESULT(cord_id, result) returns the result of the computation to the conductor.

`cord_id` is the conductor-provided identifier (see `CORD` below) for the cord this result corresponds to.

`result` is the result value itself.

`CORDDIED(cord_id)` notifies the conductor that the cord has terminated without a result (e.g., the user decided to kill it). This message is sent instead of `RESULT`.

`cord_id` is the conductor-provided identifier (see `CORD` below) for the cord that died.

A.2.2 Messages sent by the conductor to the worker

`CORD(Some (cord_id, program_id, arguments))` is sent in response to `GETCORD` if the conductor believes it has a cord the worker can verify and run.

`cord_id` is a unique identifier generated by the conductor using the program code and the arguments to it. Thus, runs of the same program with different arguments have different `cord_id`s.

`program_id` is a unique identifier generated by the conductor using the program code. It is useful for avoiding sending code over the network when it is not necessary to do so — i.e., the worker already has a copy of the code locally.

`arguments` represents the arguments to the code the conductor wants the worker to run.

`CORD(None)` is the response to `GETCORD` if the conductor has no suitable cords available. After this message the conductor-worker conversation ends.

`PROGRAM(p)` is the response to `GETPROGRAM` and provides the code the worker needs.

`p` contains all the necessary code packed into one string.

`CANCEL(cord_id)` tells the worker not to bother completing the cord it is currently running. If the worker sends back a result regardless, the conductor may choose to ignore it.

`cord_id` identifies what cord to cancel; it was first provided to the worker in the `CORD` message.

A.3 Submission and querying protocol

This protocol is used locally to submit cords and to query them for results.

A.3.1 Messages sent by the user or cord

`SUBMIT(program_name, arguments, dependencies)` tells ConCert to pick up a program from the local filesystem and have the conductor schedule it to be run. It elicits `SUBMITTED` response if all goes well, or alternately a `SUBMIT_FAILED`.

`program_name` is the string identifying the program; files containing the code will be located in a directory known to ConCert and will have TAL-related suffixes (.tal, .tali).

`arguments` is a string representing the arguments to be used with the program being submitted.

`dependencies` identifies any other cords on which this cord depends (i.e., whose results it needs).

`QUERY(cord_id)` requests the result or state of a cord.

`cord_id` is the identifier supplied by the `SUBMIT` → `SUBMITTED` exchange.

A.3.2 Responses from ConCert

`SUBMITTED(cord_id)` tells the submitter that the cord was successfully submitted.

`cord_id` is the identifier ConCert returns for use as a handle for querying.

`SUBMIT_FAILED(error)` tells the submitter that submission failed (e.g., ConCert could not find the program files).

`error` is the error message string.

`WAITING` is one possible response to a `QUERY`; it means that the cord being queried is waiting for a result from at least one cord on which it depends.

`QUEUED` is another response to `QUERY`: the cord's dependencies are all filled, and it is available for workers to run.

`RUNNING` is a third `QUERY` response: the cord is currently being run by a worker.

`FINISHED(result)` is the response to a `QUERY` that means that the cord has been completed.

`result` is the cord's result.

`QUERY_FAILED(error)` tells the submitter that querying failed (e.g., a nonexistent `cord_id` was given).

`error` is the error message string.