Using the Isis^2 Library to Build Scalable High-Assurance Services.

Ken Birman; http://www.cs.cornell.edu/ken; ken@cs.cornell.edu; 607-255-9199.

Introduction

The need for high assurance has never been greater: with the trends towards data centers of all sizes and shapes (ranging from small racks of just a dozen or two machines to massive cloud computing data centers with hundreds of thousands of them), developers of modern computing systems need to target the Web, employ Web Services APIs, and yet somehow ensure that the solutions they build can scale out without loss of assurance properties such as data security (who knows what the other users of the cloud might be doing… or what might be watching?), consistency and fault-tolerance.

Isis^2 does this by offering you a single new "abstraction" that is easy to use and remarkably powerful: the "object group", in which a set of executing programs each has an instance of some object (identically defined), and when the programs are running, the objects are linked together into a kind of distributed object. The distributed object is much like any other object: it has private data, methods for performing operations on it, read-only and update actions, etc. What makes it distributed is that the object instances within the programs that use it coordinate, using Isis^2, to make sure that any update is applied to all the copies, that reads return a correct result, etc. Thus one writes code that uses object orientation in a normal way, but by having the object instances linked through Isis^2 we obtain a distributed, coordinated behavior.

Object groups can be small or large. A small group could be used to replicate data across some set of nodes where your programs are running, for example to create a fault-tolerant service, or to coordinate actions among a set of replicas that perform some role within your application. You can create many of these kinds of small replica groups if you like, so one could imagine using a group for each shard in a large sharded data store.

A large group might span hundreds, thousands or even tens of thousands of process instances and could be used to support a variety of big-data programming models. For this style of use, Isis^2 hosts a built-in fault-tolerant distributed hash table (DHT) and is integrated with the powerful Linq data query model within C#, and with a built-in aggregation mechanism unique to the Isis^2 system. One can build completely new styles of big-data analysis solutions this way, unlike anything that exists commercially… and do so easily, and with confidence that the solution will be fault-tolerant (self-healing), consistent, stable and secure.

An example of the guarantees you obtain this way is this: if you build a large key-value store as an Isis^2 DHT, and use the consistent versions of the Put and Query operations, you can obtain large-scale results that are computed along a consistent cut (hence totally ordered relative to Puts, even if they updated many key-value pairs all at once), and in which each key-value pair contributes exactly once to the result. Moreover, you are guaranteed that the "work" to perform these tasks only occurs at a single representative of each of the DHT replicas for a given key (a single shard representative, that is), and only involves shards that were accessed by the query. These are very powerful scalability guarantees unlike anything you'll have seen with other technologies, and they are easily achieved with the Isis^2 DHT. On top of this, the DHT is secure and fault-tolerant.

A second example of the benefits of consistency arises when using something we call the Isis^2 "out of band" file transfer infrastructure. This layer focuses on applications that maintain large data sets as memory-mapped files (such files might also be associated with a persistent log or disk representation, but Isis^2 focuses on the memory-mapped representation). With the Isis^2 OOB layer you can efficiently manage and modify the file replication patterns within the members of a group at very high speeds, shifting data, adding replicas, or deleting them, as frequently and concurrently as you wish. The OOB layer is designed to handle large numbers of OOB files (thousands if you wish), and used highly efficient asynchronous messaging to transfer the data, exploiting multicast if permitted. It then automatically checks for any chunks that may have been dropped and resends those, thereby ensuring that every recipient receives exactly one copy of each file. Last, it enciphers the data (if desired) using AES-256, so that eavesdroppers would not learn anything about their contents. Here, consistency involves such guarantees as the assurance that rereplication commands will be carried out fully, that each mapped copy will be identical to the others (of course if you modify a copy, you must take steps to replicate the updates; that won't be automatic), that the files are secured correctly, that the system won't leak data, and even that the transfers themselves make efficient use and maximally concurrent use of IP multicast without risk of overloading the network or overwhelming the network interface cards of the participants.

In this tutorial we'll see how you can exploit these and other Isis^2 tools, by leveraging object groups to replicate data (in-memory or in files), to replicate other kinds of runtime state, to coordinate actions, and to exploit massive parallelism. A group can also be used to replicate external services like databases. And you can use groups to build fault-tolerant, secure, services to do things for external clients. Isis^2 is built for the cloud: all of this will work easily in settings like Amazon's EC2 or Microsoft's Azure. The system works very well on HPC clusters too, and can even run in WAN networks, although we haven't optimized for that last case.

"Big Data" is the hot topic of the day. With Isis^2, you'll find it easy to create online services (in the sense of "interactive, with dynamically evolving data") that scale to huge numbers of members, are secure and fault-tolerant and self-healing, and that integrate very cleanly with data manipulation primitives such as the ones in Microsoft's powerful Linq library. This combination enables a completely new kind of interactive large-scale data service that marshals the power of numbers to rapidly and consistently respond to queries that demand enormous processing power.

The kinds of examples we're exploring involve situational awareness for vehicles in traffic: one represents the state of the application as a huge collection of key-value pairs spread over a massive DHT, at which point queries can track, for each vehicle, the K most important entities for it to "pay attention to" in its environment. As people and vehicles move, the car itself advances, etc, the query would just be iteratively recomputed – and the DHT can sustain such high loads that many cars can share it at a time. Moreover, you can't end up with a stutter effect, in which information for some single entity is reported multiple times, pushing important data for other entities out of the top-K list. Problems of that sort are common with architectures lacking consistency guarantees: they can easily get fooled at count the same tuple multiple times, especially when data evolves.

The Isis^2 library was built to match to the style of development used for standard object-oriented applications that use GUI builders, so that anyone comfortable with object-oriented code development should find the system easy to use. This user’s manual will review the main functionality of the system in a tutorial style that doesn’t assume very much. A second useful form of documentation is the Isis^2 API documentation. This “compiled html” document gives you simple basic information about the available methods in the API and how to call them from your code.

Today, the majority of current users are working with Isis^2 from C# on Mono (Linux) or .NET (Windows), and the system can even be used on Mono for Android (from Xamarin). A second group of users has been focused on Python, and specially IronPython, and we recently added support C++/CLI. But in fact the system can already be used from any .NET language that can do calls to C#, including the .NET versions of C++ and VB. During 2013 we intend to offer a server configuration in which Isis^2 would be accessed via local procedure calls. At that point we’ll be able to support user who prefer to work in unmanagwed C++, Java, C, Perl or even Javascript!

An Example of How Isis^2 Can Fit Into a Distributed Computing Task

You don't need to understand a lot about distributed computing to work with the Isis^2 system. Our goal is to make life easy for developers with normal skill sets and who don't happen to have PhD degrees in distributed computing to create highly assured, scalable, strongly consistent distributed services that can run on the cloud right out of the box. Of course, cloud computing and reliable distributed systems will never be quite as easy as building a non-distributed "Hello world" application in your favorite programming language, but we've done everything we can to reduce the bar by moving annoying, complex mechanisms into our library, and leaving you with the fun part.

No matter what language you plan to develop your application in, you'll think of Isis^2 in a single standard way. Isis^2 is centered on a basic, simple building block, namely the object groups mentioned earlier. Imagine that you've been asked to create new service (perhaps, it tracks the inventory of items in a big online store). For the sake of our example, let's assume that you already know enough about "Web Services" to create a non-distributed version of this service: the task is really pretty simple; it entails designing an API that can be accessed over the web, and then implementing the methods that query and update the inventory database[1].

The basic idea is to employ Isis^2 object groups as containers for state (data) that you want to replicate across the instances of your service. Isis^2 automates a tremendous variety of the tasks involved in implementing this model. For example, with our trivial inventory service example, the data we might wish to replicate is the inventory itself. With Isis^2 you can create an object representing the inventory state, and then linked these objects across the service instances that are running so as to:

1. Arrange for new instances of your service to be initialized with the current inventory.
2. Make sure that any updates are applied in all replicas, in the same order.
3. Load-balance queries across the full set, giving you impressive performance for read-only operations. Obviously, updates have more work to do and this will slow them down, but even updates run faster in the Isis^2 approach than you might have believed possible.
4. Checkpoint itself into a secure disk storage area so that if the service shuts down entirely and then restarts, it can restart into the identical state.

Some applications just have one form of replicated data: perhaps, a single object class that you use to represent the important state. (We would think of this as one "object" containing multiple "items" rather than thinking of each object as a separate object group; this avoids needing vast numbers of distributed object groups, and cuts down on overheads. Other applications might need multiple distributed object groups, each handling different kinds of data and perhaps each with its own replication pattern – its own list of *members*.

[1] A simple example of using Isis^2 to build such an application on Microsoft Azure can be seen in Appendix 1. In that example, we show that you can construct a client program that works with the standard Web APIs used for cloud computing that talks to an application running on Azure (one of the main cloud platforms), all in the standard way. This application is actually accessed through a kind of web page that has logic associated with it. And that logic, in the example, is coded in C# and can access the Isis^2 library. This allows the application implementing the web service to replicate data and coordinate actions, even though it runs in the world's most tricky environment (the first tier of the cloud is not a place one normally would think of as suitable for replicating data).

We've emphasized that you'll program in a standard object-oriented way, and yet we've described a variety of very non-standard distributed behaviors. The basic idea is to embed calls from your object methods to Isis^2 that enable Isis^2 to step in and help with non-trivial distributed actions. For example, suppose that an update occurs. In your object, you would have some form of update or "setter" method by which the object instance at which the update originated first learned of the action to take. Normally, such a method just updates the data of the object. When building a distributed object – a distributed inventory – your setter method would instead invoke an Isis^2 operation called OrderedSend that replicates the action across all the members of the distributed group: each member can thus apply the same update action, in the same order. Even the member where the action originated will do the update in the identical manner to all the other replicas.

Thus by splitting your code into the "initiation" of the action (namely the update method that invoked OrderedSend) and the "performing" of the action (namely the code to actually perform the update) we'll be able to have a single place where an action is initiated, and yet ensure that the action is carried out by all the object replicas, in the same order, securely and fault-tolerantly.

Here are some of the basics:

- You'll code your application in an object oriented language, namely C# or Python or perhaps C++/CLI. As mentioned, down the road there will be more and more options (for languages like C, we'll end up "emulating" the object orientation aspects).
- Once things are looking solid, you'll create Isis^2 groups for each of the objects that has data you want to replicate.
- You'll code some very simple event handlers, which Isis^2 calls when events such as new updates or membership changes occur. Later you may make them fancier, but at first, simple ones will definitely suffice.
- You'll code a procedure to create, and to load, a checkpoint of your group's state; Isis^2 will call it as needed (to initialize a new member, or to update the external checkpoint we maintain for *persistent* groups).
- You add some boilerplate code to tell Isis^2 to initialize itself, and… voila!

Many of our examples require as little as 20 or 30 lines of Isis^2 code to turn a non-distributed service into one that can be replicated as ambitiously as you like. And these lines of code are quite normal. The examples given here are all standard C#. Readers who know Java[2] and haven't seen C# will probably think that the look like Java but with a few syntax errors: the languages are very similar. Our online documentation shows the C++ APIs, and once we start to support languages like C and Python, we'll extend it with API documentation for those too. Eventually we'll update this tutorial to have all of our examples in all of these languages, side-by-side, using tabs to let you select the language you prefer to see our stuff in. But for now, C# will be our focus.

Let's return to our example: a replicated inventory object. An inventory behaves like a database; one would typically want to "read" from it at any single replica, whereas updates would map to multicast: a 1-all sending pattern that might take an item identifier and a change in the inventory as

[2] C# and Java are actually nearly identical in many ways, but C# has a different runtime environment, and also has some language features that Java either lacks, or presents in slightly different ways. Thus, if you know Java, our examples should look very natural, but you might still want to read the associated discussion.

arguments. Our scheme makes this look like a parallel upcall to an update method you define, with the arguments showing up much as arguments are delivered to any C# method.

How would the reads get "load balanced" over the inventory group? The answer here depends very strongly on the way that you actually deploy the service and the manner by which client systems talk to it.

If your service members each have a GUI of their own and are used directly by human end-users, the answer is obvious: when a person using the service clicks the lookup button on the GUI, your application would read the inventory, and when they click update, you would perform an update. But of course few modern applications are built this way.

More common would be to implement a "client-server" structure: the inventory would be part of some form of service, that runs on behalf of clients which are actually implemented as separate programs and probably run on distinct machines, accessing the servers over the Internet. Here one needs to work with some standard client-server package that supports load balancing, and there are many to chose from. Within the .NET framework, for example, many developers use an API called REST. The web standards have evolved into a Web Services infrastructure that is very widely employed; here the client systems are often web browsers and the web service side might be a database platform (like Apache, or Oracle) or it might be a service that directly talks to clients. For such cases one normally uses prebuilt tools integrated with your application development environment: Visual Studio, Eclipse, etc. Those tools automate many of the needed steps, such as the ones by which requests are encoded into HTML messages (in SOAP format) and sent to the service. Isis^2 doesn't "change" these aspects in any way. You'll need to work with standard solutions.

Load balancing for these client-server approaches normally is done by the cluster or the cloud platform that runs the server instances. As client requests arrive, they are automatically directed to the least-loaded of the available server instances. Again, Isis^2 doesn't involve itself in these steps: from the point of view of our little inventory service, the replication task arises at a later stage of computing, when your server instance has the request in hand, and performs an action on the replicated inventory object: the "front end" for the object group. (Later we'll see that Isis^2 has a client-server API of its own, but by the time we look at this closely, we'll also understand that it is for use internally within a set of programs that are already working with Isis^2, and that the clients aren't really external human beings; the Isis^2 client API is for quite a different case in which programs are using Isis^2 fairly aggressively and become one-another's "clients").

Thus, we have a kind of three tier architecture in mind, and Isis^2 plays roles mostly (or only) in the inner-most tier. The first-tier, as in any three tier system, is the application that talks to the human client. We don't change that in any way. The second tier is the place at which requests arrive in the data center (perhaps, an ASP.NET page). Generally, we won't use Isis^2 here, either. We'll leave the existing infrastructure in place, mapping client requests to tier-two server instances (confusingly, these are often called the "first" tier of the cloud, even though they are obviously the "second" tier in a standard three-tier architecture). Isis^2 enters the picture only in the next and inner-most layer, when the tier-two component (or first-tier cloud component, if you prefer that nomenclature) begins processing the request and talks to an instance of our replicated service. Now we've finally encountered a program you might have coded in C#, Python or C++/CLI, and this is the layer at which Isis^2 plays roles.

Appendix A explains how to install Isis^2 on the Amazon EC2 cloud platform, or other Eucalyptus-based cloud-computing infrastructures. This isn't a requirement: Isis^2 can also be used for applications running directly on a cluster of machines that you manage by hand, but it does offer a way to access really large numbers of servers.

This now raises a question: are there ways to take advantage of replication to do more than spread the read queries over lots of replicas? If we were to rent lots of nodes on EC2, would that pay off in other ways? In our example as outlined above, read requests are handled by any single server instance, but updates reach all server instances. As it turns out, you can also use Isis^2 for fancier scaled-out behaviors. For example, the system allows you to send a Query to the whole group (we've capitalized Query in this sentence to help you start to think in terms of the Isis^2 API, which calls this particular operation by that name), so that each of its members can contribute part of the response. By doing so you can marshal massive parallelism, writing code in a way that gives you an N-fold speedup with N members in the group, and perhaps can scale to huge values of N (we say "perhaps" because getting this to work as efficiently as possible does take a little bit of testing, tuning and some advance planning as well). With this approach the user can send in a question, and you'll be able to put 10 processes…. or 10,000 of them… to work in parallel.

Isis^2 provides strong *consistency* guarantees. This means that any replica will see the same sequence as any other, every member of your system will know it's "role" (we number the replicas, from rank 0 to N-1, and every replica knows its own rank, the value of N, and the rank of each other replica). Thus, you can build a system in which each replica plays a distinct role and yet the add up to a total story. For example, with 10,000 replicas you could search a 25000 page telephone white pages directory to do a reverse phone-number lookup, and each replica would do precisely 2.5 pages of searching. Not a single phone-book entry would get missed, or searched twice. Moreover, and this is the part that can seem a bit mind-bending, we can provide a meaningful guarantee even if the data is changing while you are reading it, and even if service replicas are joining or leaving or failing while the request runs! Would a telephone-book search have all of those issues? Perhaps not. But some systems do have all of them (think of a system controlling the smart power grid) and our intent is to enable the easy cases, but also the hard ones.

And again, even though the model is fancy, the logic you code to implement these behaviors can be as little as a line or two of very standard-looking C# code (or whatever language you use).

All of this makes it easy to implement various fault-tolerance behaviors; we'll show you how. And you can also secure a group so that snoopers, watching traffic on the wire, won't be able to decipher a single byte of application data.

Every system has its limits, and Isis^2 is no exception. For example, the system isn't designed to tolerate arbitrary "Byzantine" failures, such as bugs that damage the memory in some replicas. Isis^2 offers several flavors of multicast, and expects you to pick the version that will be fastest but still correct for your purposes. We'll explain how to do this, but the job of making that choice will be yours.

We should mention few other useful things that you'll want to know about. One is that Isis^2 automates the creation of *messages* so that you can work entirely with variables and procedure calls. Thus, if you want to send an update identified by a string (maybe a product name) and that sets a new price (a float), rather than worrying about the external representation of these kinds of data objects in messages, you'll just call Isis^2 system calls providing the name, and the price, and we'll later call back to your update handler with the name and price as arguments. If you query a group, and several members respond, each sending back a different answer (perhaps, a string), we'll return a List of strings to you (and of course, you can use any data type you like, although you do need to register any new objects that the application plans to send as arguments or receive in replies, so that Isis^2 will know how to create messages from them, and how to create new instances when they are received in messages). The same approach is used to create checkpoints. In typical applications, you'll never need to see the Msg objects used internally by the platform at all.

This does mean that Isis^2 will need to know about the types of data you plan to ship around. We support most of the obvious standard types, including .NET Arrays, Lists and KeyValuePairs, but for fancier things you'll need to provide some help, as we'll explain below. For example, if you invent a new data type that has various fields, you'll be able to include objects of that type into your distributed applications but will first need to declare some information about the objects, so that Isis^2 knows how they look and how to put them into messages, and extract them back out. For languages like Python that have very flexible notions of types, you will need to tell us which types are legal for the methods you define, so that Isis^2 can match incoming messages to the appropriate handlers. (Yes, we know that in Python, one doesn't really have to do this at all, but the point is that Isis^2 also has to work for languages that are very rigid about type matching, and we're trying to support different styles of use all at once).

Similarly, Isis^2 automates startup: each application process automatically figures out its own Address. An Isis^2 Address is an object containing an IP address together with process-id numbers and other data, and automatically finds the Isis^2 rendezvous service, which we call the Oracle. For most purposes, you just launch your application and it will join itself to the running Isis^2 system without any special work on your part.

Finally, Isis^2 tells you a lot about membership in process groups that your application joins. It does this by means of an upcall to a method you'll supply, with a *group view* as an argument. The group view lists the members of the group, the most recent update to the list (joining and departing processes), the rank of your application process in this list, etc.

Thus, your job is just to write a few methods, register them, and then Isis^2 can do most of the rest of the work. Most developers wrap the Isis^2 replicated objects within their application in application objects, hence the user's of those objects see completely standard object-oriented interfaces and can be almost completely unaware that they are really just front-ends to a fancy distributed functionality.

Figure 1, about two pages from here, shows all of this in a pictorial form, with time advancing from left to right, the time-lines for various processes shown as the events we've discussed occur within them, and big blue ovals denoting the new views of the process group this application is using (a checkpoint used to initialize a joining member is seen as a white arrow inside the first of these blue ovals: some active group member checkpoints its state, sending it to a joining member). Of course a real system could use many groups, not necessarily just one.

Isis^2-Hosted Distributed Hash Tables

A second powerful way to use Isis^2 groups involves the construction of a distributed index spanning the members. Called a *distributed hash table* these indices are (key,value) sets in which the key is used to spread the (key,value) tuples over the members of the group, using a form of hashing that will spread items in a very uniform way. For fault-tolerance, items are replicated and you can specify the replication factor. For example, with 1000 members in a group, and a replication factor of 2, each group member would have 1/500th of the total set of tuples. Get operations are spread over the replicas, while Put operations update the full set of replicas. The keys and values can be of any type you like, as long as Isis^2 understands how to marshal that type into a byte-serialized format.

In the most common mode of operation, an Isis^2 DHT will be slightly less reliable than a full virtually synchronous group, because each (key,value) pair is only replicated to a small subset of the group members and because we don't use virtually synchronous multicasts to perform the DHTPut operation. Thus one can witness a partially completed Put and, if all members in some partition fail simultaneously, the portion of DHT contents they owned would be lost. But this can be made very unlikely. The big advantage is that huge amounts of data can be managed in this manner.

You can force the DHT system to perform Put operations using a stronger method if you wish to do so: DHTOrderedPut, like Put, adds new (key,value) pairs to the DHT, but unlike the normal Put, takes a list of tuples which are atomically added, and does so using an OrderedSend to the full group. DHTOrderedRemove, similarly, will atomically remove a set of (key,value) pairs given a list of keys as its argument. These however are more costly than the basic DHTPut.

When a collision occurs (same key, but a new value), the default behavior for the system is to discard the value associated with the earlier Put and retain the value from the one done more recently. But you can implement very different behaviors by calling DHTSetPutCollisionResolver() and specifying one or more methods that each take a key and two values (the old and the new), and return the value you prefer that Isis^2 retain for the corresponding key and value types. You could combine the values in some way, discard the old, or even decide between them on the basis of some special approach of your own. Notice that this allows you to create and maintain lists of values associated with a single key. For maximum flexibility, we use type "object" in this API, but keep in mind that you'll typically want a single value "type" for any given key type used in DHTPut. Thus if you do want to maintain a list of values, we suggest that you use List<someType> even in the DHTPut itself. Otherwise, you'll end up with the DHT containing a mixture of key,values that are of form KeyType,ValueType and others of form KeyType,List<ValueType>. You can certainly do this, but it leads to confusing code. If you don't specify type arguments, the method you supply will "match" all possible key and value types; if you do specify them, the method will only be called for keys and values that match.

In addition to the basic Put and Get API, the Isis^2 DHT offers another powerful option: the ability to have group members perform parallel queries over the DHT elements. To exploit this feature, the developer spreads data within the DHT, then uses the Isis^2 Send, OrderedSend, Query, or

OrderedQuery operation to send a request to the full group membership. (The ordered versions guarantee consistency). The members pull up their local portion of the (key,value) list using the g.DHT<KT,VT> API, which returns a read-only list of (key,value) pairs matching the given types. One can then use the .NET Linq feature to perform arbitrary database-style queries on the extracted data, and finally combine the results. For example, given (key,value) pairs in which the keys are employee-id numbers and the values are salaries, a member of the group could form a histogram of its "share" of the tuples this way: g.DHT<int,int>().GroupBy(kvp => kvp.Value/1000);

One thing to be aware of is that modifications you make to g.DHT won't impact the underlying DHT itself. Obviously, your code can issue new DHTPut or DHTRemove operations at any time, and these certainly will change the contents of the actual DHT. But direct changes to g.DHT are discarded. The simple way to understand why Isis^2 adopts this rule is to remember that (key,value) tuples are generally replicated onto multiple processes. However, the g.DHT action is performed *locally* by some process with a portion of the full DHT, or perhaps executed in parallel by some set of processes each of which is holds a partition of the DHT. Thus if we allowed updates to be performed directly on the DHT through the g.DHT API, it would be very easy to leave the DHT in an inconsistent state.

Obviously, because data is spread within the group, no member computes the full answer here. Instead, each computes a "contribution" towards the desired result (in this example, each has its own count of how many employees fell into its portion of the DHT, broken down into bins by $1000 salary increments). The computation is inherently redundant: if the DHT is replicated R times, there will be R duplicated responses for each partition of the data (or fewer if the group temporarily has lost some members, resulting in a brief reduction of the number of members in some partition). But Isis^2 also has an automated mechanism that eliminates this duplicative computing. The mechanism is tied to something called a QueryKey object and is explained below.

When using DHTOrderedPut to add (key,value) tuples to your DHT, it becomes possible to use OrderedSend or OrderedQuery to initiate these kinds of parallel DHT search or update operations. Here you gain the full power of virtual synchrony: all members will see consistent DHT contents and hence the responses to your operations will be coherent across the group; in effect, they occur on a consistent cut. The downside is that DHTOrderedPut is slower than DHTPut, but this overhead vanishes when large numbers of (key,value) operations are initiated by a single DHTOrderedPut. DHTOrderedGet takes a list of keys and returns a list of (key,value) pairs, with one value for each key that was found within the DHT. The operation is strongly consistent, representing the result of a parallel DHTGet performed along a consistent cut across the DHT.

To identify a query of this sort, we've introduced a new "QueryKey<KT>" object: you supply a list of keys or type KT when you construct it, and then supply the query key object both to your group members, who use it when reporting their aggregation results, and at the initiator, who waits for the result using the query key as an identifier. The QueryKey object you create for this purpose should appear immediately after the request code in the Send or OrderedSend if you wish to use this feature. As just mentioned, when the QueryKey feature is used, computation will *not* be redundant: although your data is replicated within shards, the QueryKey mechanism picks a single representative per shard, and only that representative participates. Thus no duplication of computing occurs.

Virtual synchrony also ensures rapid adaptation if members fail, and the Isis^2 DHT allows you to preposition a few “spare” processes (i.e. your group might have 1003 members rather than 1000), in which case a spare will be slotted in to replace any member that fails, minimizing the amount of remapping needed after a crash. Thus virtual synchrony offers a route to very strong consistency (but an optional one), makes the solution scalable, is rapidly adaptive after failures, and secure. We don’t know of any DHT that offers stronger properties.

Getting Started

Using Isis^2 involves three simple steps, and this tutorial will walk you through them. In order to write a program that uses the library, you'll first need to *install* the Isis^2 library. We provide it in source form under a free BSD license (the standard 3-clause version). You'll download the source file from isis2.codeplex.com, compile the source using the C# compiler, and link it to your application. Although the system is open source, we don't recommend that you change it: this is not the kind of open source where ten-thousand developers add features. For the time being, the source is offered to assist you in debugging your application, and for communicating with the author, who (which his students) is providing support (free) for the user community. When the system is sufficiently stable, we may switch to a more standard open source-form model in which community bug-fixing would be possible, but right now, you'll compile the system and run it… we'll fix any bugs.

On Visual Studio for Windows, for example, the easiest approach is this: Open a new Visual Studio "console" project (a console project is a program that expects to run from a shell command, like on Unix, and in fact works just as on Unix, with a "main" method that gets passed the arguments, etc). Then you can just add Isis^2 as an additional "file". In this approach Isis^2 is just a part of your project.

A more common approach is to create a new "C# Library" project using the Isis^2 source. Compile it and then add a "reference" (go to the "project" window of your console application, click "add reference") to the Isis^2 Library dll you created in the first step. This way the dll can be shared among multiple users and if you download a new version, all of them will be able to benefit from the upgrade. From Linux, where you might be working at a command-line level to do code development, or using Eclipse, the process is similar except that you'll use the Mono C# compiler (monocs) to create the library file.

Once you have Isis^2 available in your program, you import the Isis^2 API in whatever way the language you prefer supports: a "using" statement in C#, an "import" in Python (but please check our special notes for Python users first), an "include" in C++/CLI, etc. You can then start to call Isis^2 primitives.

Although you could use Isis^2 without learning more, we recommend that before trying to run your first application, you learn a little about about the *virtual synchrony* model that Isis^2 employs; this tutorial covers that in the pages that follow. Virtual synchrony is the key to using Isis^2 successfully, but unlike some distributed systems ideas you may have encountered, this is an especially easily and intuitive one to work with. Don't let the name scare you off – it just means that your system will *look* (hence "virtually") as if *one thing happens at a time* (hence, "synchronous").

Finally, you'll write applications in a new style that fits well with the way cloud computing platforms manage your software. We talked about this in the previous section but will see more examples in the pages that follow. The resulting system can then be deployed onto any network with a few PC's under your control, whether those belong to you and your colleagues in the office, are rented from a company like Amazon through its EC2 service, or are deployed into a major full-scale cloud setting where you might run the solution on huge numbers of machines. The benefit of Isis^2, compared to other ways of building cloud applications, is that it is compatible with the most widely popular platforms, such as Azure, but offers you powerful additional capabilities such as fault-tolerance, consistency, data replication, security, etc. Thus by combining Isis^2 with your favorite cloud platform tools, you put those tools on steroids!

Virtual Synchrony: the Secret to Isis^2

When use use the Isis^2 system is that your code runs in a new kind of managed framework. You are no doubt used to managed frameworks like .NET and the Java JVM: they offer support for threads, for memory management, etc. Some frameworks go further: tools like Azure, Google AppEngine and Hadoop (aka MapReduce) offer higher level managed frameworks aimed at making cloud application development easier. Microsoft's Dryad platform goes even further, offering a kind of distributed programming language that "compiles" to infrastructures such as Hadoop.

Isis^2 is a cousin of these kinds of systems, but starts by introducing two features that few existing systems offer: an execution model, and a new kind of distributed resource management structure.

The execution model is called *virtual synchrony* and dates back almost 25 years to work that started around 1985 at Cornell and led to the first Isis system, something we called "Resilient Objects in the Isis System" and later reimplemented as the "Isis Toolkit". Call those Isis^0 and Isis^1. The Isis flavor of virtual synchrony was used to build the French Air Traffic Control System (now deployed widely in Europe), the US Navy AEGIS warship, and even ran the New York Stock Exchange for more than a decade. In all of these systems, as in many other Isis Toolkit applications, routine crashes and similar events won't bring down the application as a whole: replication is used to ensure that the system can survive the failure of some small number of its components. This is why the various crashes that happened over the decade that the NYSE ran Isis never brought that equity trading environment: it was "self healing." With Isis^2 your software can draw on the same core ideas.

The Isis Toolkit wasn't the only system to use virtual synchrony. Perhaps you've heard of JGroups (part of JBoss). That system was developed as part of work done at Cornell to create a successor to Isis^1 (the successor came in two flavors, one called Horus and one called Ensemble, and Ensemble was later recoded into C (C-Ensemble) and Java (JGroups)). IBM used virtual synchrony in its Web Sphere product line for Web Services. A widely popular platform called Spread was used extensively for management of smaller data center services. And virtual synchrony is very closely related to the Paxos protocol suite (in fact, the model we implement in Isis^2 is really a fusion of the State Machine Replication model supported by Paxos and used in Google's Chubby system, with the older Isis^1 version of virtual synchrony). So if you are a fan of Google's Chubby, Isis^2 should feel familiar in many ways. Same goes for people who love Yahoo!'s Zookeeper service: that uses a version of virtual synchrony, too. You can read more about the history of this area in a volume from Springer Verlag called "Replication: Theory and Practice" and available in the LNCS series:

> **History of the Virtual Synchrony Replication Model.** Ken Birman. Chapter 11 in Replication:Theory and Practice. B. Charron-Bost, F. Pedone, A. Schiper (Eds) Springer Verlag, 2010. Replication, LNCS 5959, pp. 91–120, 2010.

The best way to understand virtual synchrony is with a picture, showing the kind of executions that can arise in a system that implements this model. Think of the execution as a form of virtual environment, in which events that occur in the real world trigger events in the virtual one, but where the runtime system schedules those events to ensure that the resulting execution looks like this sort of picture. As you'll see below, time runs from left to right, and there are execution "timelines" for each of a set of application processes running on the nodes in some kind of data center or cluster. They send each other messages, which are the diagonal lines running between the execution timelines.

Rather than present virtual synchrony with just a single execution picture, it will be convenient to show three tightly related ones, which is what we've done here. The first of these is what we call a "reference execution" (on top), and it looks simplest too: just a single time-line with a few events occurring along it. The timeline shows a server process and the events presumably occur in response to requests from clients, although those clients aren't shown. Next, we see a "synchronous" execution (on the lower left). A second and perhaps more familiar term for this style of execution is State Machine Replication. Finally, on the lower right, we see a "virtually synchronous" execution. Again, keep in mind that we've simplified these figures by leaving client systems off, but that they would also have timelines, and should be understood to be interacting with the processes making up our system from time to time, triggering the events depicted here.

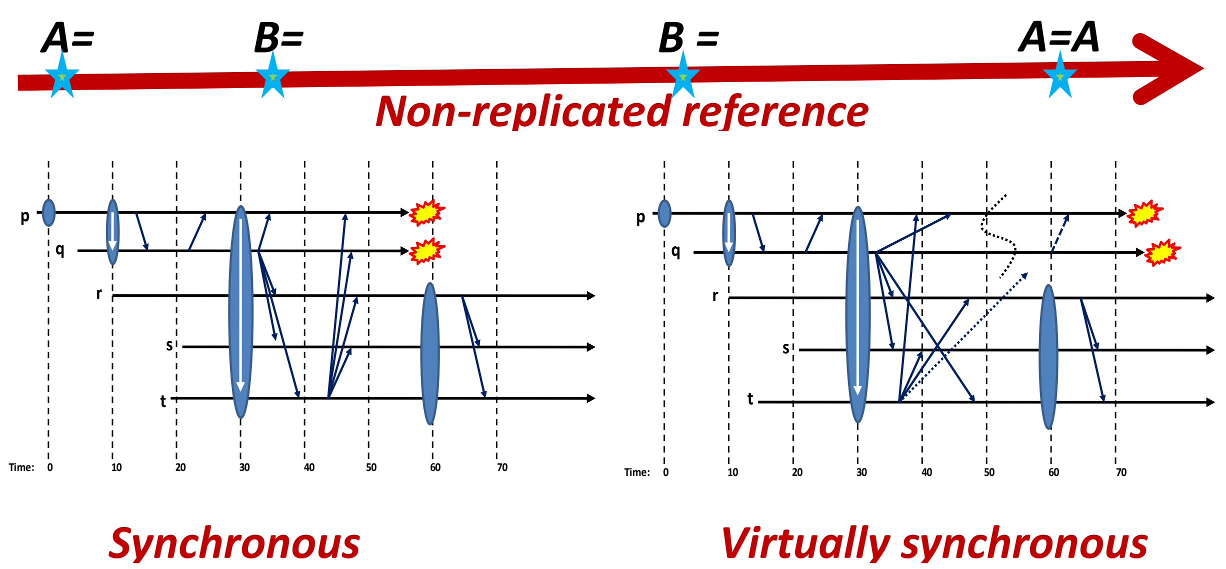


Figure 1: Isis^2 employs the powerful virtual synchrony model, in a new form that actually combines virtual synchrony with a variant form of the Paxos State Machine Replication protocols. In the fastest configuration, all updates and multi-process queries are performed with a single IP multicast per request.

The Reference Execution

What's happening in these figures? We want you to imagine an application that has some sort of object within it, managing a collection of data items that we've named using symbols. Here we're going to pretend that there is a single object and it manages two data items, A and B. Sometimes users do operates that change the values of A, or B, or both: we see A set to 3, then later B is set to 7, etc. The object could be a database or file, but could also be some sort of C#, C++ or Java class that you've coded, in which case A and B might be instances of that class.

In C# you could implement a class that would let you create this sort of object in about six lines of code. Nothing non-standard or fancy intended.

We've depicted our reference execution as a sequence of operations, but as you'll see below, this isn't really a requirement. The reference run is really any "correct behavior" that you can dream up, in a non-distributed, failure-free setting. So you can certainly plan to use threads (which would give you one execution timeline per thread), asynchronous file I/O, etc. The key insight is just that the reference execution defines correctness for us. Our goal as distributed programs is to trick our users into thinking that the distributed system we've created isn't distributed after all: it behaved just like the non-distributed one, so (as they used to say about a men's hair coloring product), "only you and your hair dresser will know for sure."

The Synchronous Execution

Now let's look at the execution on the bottom left: the synchronous one. The idea here is that processes p, q, r, s and t are "cooperating" to mimic the non-replicated reference execution so that even though the work of the object is now shared by multiple processes, which could be running on multiple computers, the "state" of the replicated object advances through exactly the same events that the non-replicated reference execution experienced. Thus, the arrows could be carrying updates or locking requests needed to prevent conflicting accesses from occurring simultaneously.

What about the blue ovals that encompass first just p, then and q (with a white arrow inside), then p,q,r,s and t? These are intended to help you visualize the idea of an object "spanning" more than one process. So in our figure, p actually created the group that implements our object. Then q joined, and the oval stretched to include both p and q, with p oldest and q a bit younger. This will matter later: we say that p has rank 0 in the new view, and q has rank 1. Next we add r, s and t, etc.

The white arrows denote the creation of a checkpoint by p (just a set of data representing the state of the object when the membership changed) so that q, and later r,s and t, can initialize themselves.

We probably should have put little stars designating the exact same events that were shown on the reference execution, but you can imagine that the first arrow, from p to q, is the update of A=3. The second from q to p is perhaps the update B=7. The next two "multicast" arrows (from q to p,r,s and t and from t to p,q,r and s) are perhaps the updates corresponding to B=B-A, and to A=A+1.

Next we see a crash: p and q leave the group, and now only r, s and t remain. Perhaps that last multicast from r (the new rank-0 process) to s and t involves some sort of fault-handling logic. Isis^2 has various ways to detect crashes: it pings nodes to check their health, and applications can also report on apparent problems if corrupted data or timeouts make it obvious that some service is malfunctioning. So that's how Isis^2 learned about the crashes in the first place: something timed out. The Isis^2 uses pinging to detect failures automatically, but these mechanisms are often extended by user-supplied failure detection logic, in which the application using the Isis^2 system might tell the system that some process has failed, perhaps because of a timeout or because data was corrupt in some detectable way. (One can get into all sorts of philosophical debates about what a failure is, or how to handle cases where p says q has failed and vice versa; Isis^2 just trusts these failure detections and if they are incorrect, the "dead" application throws an exception).

Things to notice: every process sees the identical events, starting when it joins, and sees them in the identical order. Joins and failures are reported: if a group member is interested, a "View" data structure listing the members will be delivered, via upcall, each time the membership changes. And even if the processes aren't interested in seeing the views, they are part of the underlying event sequence. Everything is totally ordered: one event happens at a time, system wide, in a closely synchronized ballet that might entail coordination across the nodes of an entire data center. Finally, joins and failures are reported in a consistent, coordinated way.

We didn't show any locking here but if we wished to do so, we could easily have used the same sort of multicasts to implement a locking layer (later in this tutorial, we'll show you how). Then only one updater at a time would have permission to update A, or B, or both.

We'll say that the job of a synchronous execution is to mimic some reference execution. We implemented our object using a group of replicas (and Isis^2 will automate most of the hard work, as you'll see in a moment) yet they behaved just like the reference service might have behaved, given the identical requests in the same order. In our figure, the reference execution was just a single sequence of events, so the synchronous execution should show a sequence of distributed events, and it does: the order in which these distributed operations occurs is mimicking some order in which events might have happened for the synchronous run. Of course, as mentioned, we used an especially simple synchronous run, but the idea is the same: the synchronous behavior should be indistinguishable from the reference behavior, if you just look at the operations performed and the results that clients saw.

This, for those who know about Leslie Lamport's work, is actually a pictorial depiction of the execution model used in his Paxos protocol, and in other implementations of State Machine Replication (SMR). SMR was a topic first introduced by Lamport, and one on which my close colleague Fred Schneider worked. Most researchers give Fred the bit of credit for making SMR famous. Google uses SMR and Paxos in their Chubby locking service, which is central to a huge variety of Google services and applications. Chubby is a coordination service, but the way it works is pretty much exactly as seen in our figure: its membership can change dynamically, and the members handle lock and unlock requests in a coordinated, identically replicated manner.

But notice also that update-intensive applications can't obtain any real speedup in this model. Everyone does every operation, in identical order. Indeed, if the updates are done from multiple processes, SMR could be slower than if there was just a single process, because it can take time to agree on the event ordering. SMR requires this property, and requires each object to be a *deterministic* state machine: meaning that the application must not use threading, access system clocks or embody other sources of unpredictability. Every copy does exactly what every other copy does, in lock-stop. SMR applications can get some speedup by sharing the query workload in a load-balanced manner over the copies, but even this can be less than one might have expected, because the fault-tolerant implementations of the SMR model normally require that queries access multiple SMR replicas in order to obtain the correct current state.

In fact the story is even more extreme. We didn't start with a fancy, multithreaded, reference run, so we started with a goal that looked completely synchronous. But many servers would be multithreaded and asynchronous, and for those, the reference run might embody quite a bit of concurrency: the ordering obligation would be a partial order, not a total (sequence) event order. But SMR forces the reference run into a total order: it can only talk about sequences of events. So there is a sense in which SMR forces us to limit our attention to a sequence reference execution. In effect, SMR tells you (as the designer) to not bother to use asynchronous, multithreaded, service designs. This means you might be starting the job with one hand tied behind your back: on modern multicore machines, a single-threaded service won't be a great performer! On the other hand, SMR does show us one way to replicate some kinds of service (namely, sequential ones), preserving correctness and guaranteeing progress to the degree that our multicast system can guarantee to deliver multicasts.

The Virtually Synchronous Execution

This context leads us to the figure on the bottom right above: the virtually synchronous execution. By now you can guess what we've done: we're relaxing that rule about lock-step execution. Virtual synchrony allows you to code up anything you like, provided that you can convince yourself that the end result is consistent with some possible closely synchronous execution. Basically: do as you like, but keep SMR in the back of your mind. But also, keep in mind the observation we made earlier, about SMR being overly constraining.

We're going to see that with virtual synchrony, we can get very high levels of parallelism and hence significant speedups relative to close synchrony (to SMR). But we can also mimic SMR in a precise, step by step way, although doing so abandons some of the speed we were after. In fact one way to understand Virtual Synchrony is that it offers state machine replication in two forms: one aimed at consistency for replicated soft-state (data that has no associated disk files or databases "outside" the system), and one aimed at replicated hard-state (such as a replicated database). The later protocol happens to be a version of Paxos, hence readers familiar with Paxos and convinced that they need to use Paxos could use Isis^2 for any purpose where Paxos would be right (use the SafeSend primitive in this case).

If your goal is to maximize performance, it will turn out that you probably don't want a true SMR implementation. In Isis^2, this performance-optimal solution will be one that uses a primitive called Send (or Query), but in opting to use it, you'll also be explicitly permitting Isis^2 to weaken the model in one specific way. Our little figure illustrates the core issue, and it may sound somewhat trivial: there is a single stray update operation, shown as an arrow from p to q, that seems to be sent after p and q have lost contact with the rest of the system (perhaps, they were victims of a *network partitioning failure*, meaning that somebody accidentally unplugged the network connecting their rack of nodes to the rest of the data center). Perhaps this message did something; maybe it set A to 88. So we'll need to understand precisely what the implications of relaxing the SMR model might be.

In a nutshell, when you ask Isis^2 to guarantee "safety", by using the SafeSend and SafeQuery primitives, you force it to run a true SMR style of protocols. These are slower but very durable in the event of crashes, and stray events like our little p→q message can't arise: Isis^2 simply won't allow it. (It implements this by delaying delivery for some messages until it is sure that the execution will be safe in the SMR sense, and this involves extra round-trips, hence is slower).

In contrast, when you let Isis^2 run with all the stops out, you get *much* faster multicast and query performance. You just use Send and Query (or OrderedSend and OrderedQuery) and Isis^2 skips those message delaying steps, delivering messages as early as it can, consistent with the ordering rule you requested (Send and Query are FIFO on a per-sender basis, while OrderedSend and OrderedQuery are totally ordered no matter who sends the requests). This is the case mentioned earlier; we use it with soft-state. And the reason is that these faster protocols have a minor catch: by delivering early, there are obscure failure cases, very unlikely, that can provoke events like the p→q multicast deliver that was never seen by any other process, and essentially was "erased" by the crash.

Your decision on whether this sort of behavior is tolerable or not will have big performance implications. The rules of thumb we recommend are these. Think first about whether the messages your system sends have a long term, durable consequence. An example of an ephemeral message is a load-balancing update. An example of a durable one is a message that updates a file or launches the moon mission. Generally speaking, the fastest Isis^2 communication primitives are suitable for things that are ephemeral, and must be used with care, or not at all, when you plan to update something durable.

But not all durable updates demand the strongest ("safe") primitives. Suppose that we can throw away that file in the event that a crash occurs: the state of the live part of the system is what matters, and we'll use state transfer to copy the live state to a joining member (or a restarting member that previously crashed). Here, the update was durable but we deliberately discarded it, making an exception perhaps for the case where the whole group crashed (Isis^2 will help with that, no matter which primitives you use). So: if state transfer can be used to initialize joining members, and they don't need to "keep" data from one period of execution to another, the fast primitives suffice.

The case that remains involves launching moon missions. For those, either use the safe primitives, or try calling the Isis^2 myGroup.Flush() or Flush(k) primitive before launching the rocket, updating the file, or dispensing the cash. Either approach will work, and will yield a true SMR execution. The distinction between g.Flush() and g.Flush(k) is that the former waits until pending multicasts are stable at all group members, while the latter waits only until pending multicasts are known to have reached *k* group members, counting the sender.

Similarly, the default configuration of SafeSend waits until a multicast has definitely reached *all* members of the group before delivering the message to *any* member (you can override this using g.SetSafeSendThreshold() to specify a threshold, Φ, of members that must have copies, in which case the sender waits until Φ-1 acknowledgments are received). There also arises a question of what it means for an update to be durable: by default, SafeSend saves data in memory, but you can use SetDurabilityMethod(new Group.DiskLogger(myGroup, logname)) to override this; we offer a pre-built disk-based option (the Group.DurabilityLogger class used here), or you can also provide a of your own. Obviously the in-memory option is fastest, but it also brings some risk: if all the acceptors fail during the first phase after acknowledging, the leader might deliver a message to some but not all receivers, and durability would be violated.

Thus, a Safe multicast runs in two phases: it sends, waits for enough confirmations, and then sends a second message that triggers delivery upcalls. This, of course, could be fairly slow, and it scales poorly in Φ (at best, SafeSend will have a linear slowdown as you increase Φ). For readers familiar with Paxos, SafeSend is exactly the same as Paxos. Φ is the size of the Paxos "acceptor" set, and the full group is the set of "learners".

And if this all seems terribly confusing, and performance doesn't even matter to you, just keep in mind that we didn't give them the name "safe" casually. You are *always safe using the SafeSend and SafeQuery operations. They may be slow, but they won't surprise you.* In contrast, a user who wants to obtain the highest possible performance can use the flexibility Isis^2 offers to obtain much more parallelism in the virtual synchrony execution than would be legal in an SMR execution. So the real choice is between speed and a subtle form of complexity.

Here's a paper explain exactly what model Isis uses. It includes citations to the other models, such as the State Machine Replication work and Paxos, mentioned above.

> Virtually Synchronous Methodology for Dynamic Service Replication. Ken Birman, Dahlia Malkhi, Robbert van Renesse. Submitted for publication. November 18, 2010. Also available as Microsoft Research TechReport MSR-2010-151.

A textbook covering this material is:

> Reliable Distributed Systems: Technologies, Web Services, and Applications. Ken Birman. Springer-Verlag. March 2005. ISBN-13: 978-0387215099

A new edition of this text is in preparation, and will be available in 2012. The new edition will focus much more on cloud computing, and has specific treatment of the Isis^2 technology.

Network Partitions

Many modern cloud computing platforms run services in each of a set of data centers. Internal to a cloud-computing data center, network partitioning faults are never visible to applications: if nodes become disconnected (e.g. a top of rack switch fails) those nodes shut down, and the large-scale network is always so redundant that if something fails, the network remains connected while the operations team fixes the faulty component. Isis^2 is often used in this situation, with the entire application running in a single data center.

Two parameters control the handling of partitions: ISIS_IGNOREPARTITIONS and ISIS_IGNORESMALLPARTITIONS.

ISIS_IGNOREPARTIONS is **false** by default. This parameter tells Isis whether or not to check for partition events. If false, then if a failure causes the Oracle to drop from size N to a size that isn't a majority of N (e.g. if N=5, 2 or smaller; if N=3, to size 1) in a single event (e.g. for the N=3 case, if 2 members seem to have failed at the same time), Isis will shut down. The idea here is that if in fact N=3 and the system splits with 2 Oracle members on one side and 1 on the other, the side with 1 will shut down.

But now consider the case where N=2: the Oracle has 2 members and one of them fails or shuts down normally. In the default mode with ISIS_IGNOREPARTITIONS=false, the remaining healthy Oracle member will also shut down! This is an annoying behavior that shows up often while debugging and we added one more parameter to suppress it.

When ISIS_IGNORESMALLPARTITIONS is set to **true**, which is the default value, if the Oracle size is 2 and drops to 1, no partition is sensed and the remaining member continues to operate normally.

You can override both behaviors. If you set ISIS_IGNOREPARTITIONS=true, the Oracle will never shut down because of a partition event. And if you leave ISIS_IGNOREPARTITIONS=false but set

ISIS_IGNORESMALLPARTITIONS=false too, the Oracle would shut down in that case of a transition from N=2 to N=1, but on the other hand, no partition could form in that case. Obviously, do this only if you have a plan to replenish lost Oracle members so that you'll keep the size of the Oracle at 3 or more.

In a future version of the system we will be adding yet another feature, under which an Oracle member who shuts down voluntarily (via Isis.Shutdown()) wouldn't trigger a partition failure detection even if this means that the Oracle size drops from N=2 to N=1. However, that feature is not yet part of the system.

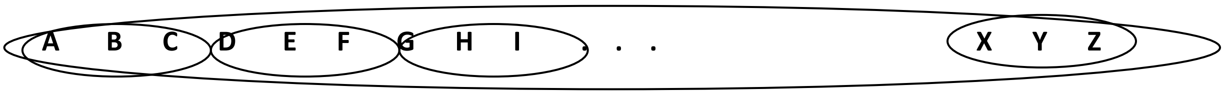
What about WAN situations, where partitioning is more common? Most applications that run in a WAN will need to use Isis^2 in a single data center at a time, then merge states by adding application-layer logic that can span across the WAN and forward any updates from one data center to the others, probably across TCP links you create, and not using Isis^2 for this at all. But nothing actually prevents you from configuring Isis^2 for use in a WAN setting. We just don't have much experience with this case and haven't optimized for it.

Sharding Data

Before getting more detailed it may be useful to just briefly contrast three replication cases, so that you'll be able to relate Isis^2 to the most popular data center service models.

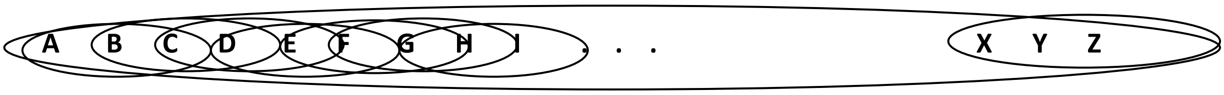
With a *single* process group, Isis^2 would normally be used for full replication of a data set and fully parallel queries or searchers, if those are sent by multicast. Some applications might replicate a data set but load-balance queries over the members, using multicasts only for updates. This works well, but isn't a style of computing you would see very often with thousands of group members. So the first model for dealing with large-scale data would be to build a single group spanning all the members, and do replication in the full group, but we don't expect that to be the most common model.

For very large groups, a more common approach is to use data *sharding,* and this is available from the Isis^2 DHT mechanisms. One picks a replication level, perhaps 3, and a rule for mapping client id's to subsets of 3 group members each, perhaps by just taking the client-id modulo the size of the group and using the member this maps to and the next two members in the group view. Then a small process group can be created for each subset of 3 replicas, like this:



Why did we end up with one group having just 2 members (Y and Z)? Because 26 doesn't divide evenly by 3: we end up with 8 groups of 3 and 1 group with the remaining 2 members.

Of course we could also have done this. Here, each process is a member of 1 to 3 shards.



Notice that our second example doesn't end up having any strange-sized shards, but on the other hand that group members have non-uniform loads, unless of course we were to "wrap" our shards around the right end and back to the left (e.g. forming a shard with members { Z, A, B }).

One could also define shards in other ways, and this leads to the puzzle mentioned above. Suppose we want our shards to be as stable as possible: when a member joins or leaves we'll want to adapt the affected shard or shards, but we want the impact of such events to be minimal. Would either of the two schemes given above work well for this purpose? What approach would you favor?

Given a sharing "rule", you can then replicate the data for a given client in the shard to which that client's identifier maps. But notice that in the event of a failure, however, these structures can sometimes be costly to reconstruct: shard membership would change (for example imagine the situation if A were to fail), and you'll need to shuffle data around to compensate (the shard that previously had {A,B,C} now would contain {B,C,D} so D will need a state transfer: perfectly feasible but a bit costly.

So what's the answer to our puzzle? One simpler and more stable approach is this. Designate one of your group members (perhaps the one with rank 0 in the membership) as the shard manager. Have it map the members to shards in such a way that once a member is assigned to some shard, it won't be removed from it again; if a shard drops below 3 members, use "new" joining members to repopulate it. With the managed group API this is quite easily done (the shard manager has a simple

API that it can use to add, remove, create or terminate groups, or lists of groups). Notice that if A was the shard manager but crashes, B will take over in this role as the new rank-0 member when the next view becomes defined. The main role of the larger encompassing group is to provide full membership data to the application and to offer a way to send multicasts to all the members when needed.

This approach, however, is more difficult to depict in a simple figure. The group that used to contain {A,B,C} will now drop to {B,C} and the shard manager will pick some additional process to add in, perhaps joining process A'. We end up with a picture that has two members on the far left and one additional member on the extreme right: hard to draw, but easy to understand and to implement.

A benefit of Isis^2 and virtual synchrony is that you can easily build solutions of this kind: they may not be trivial to illustrate in a picture like the ones above, but they often are surprisingly easy to design and implement. A further strength is that you'll have absolute certainty that every member of every shard has the correct, current data. Of course you would need to do locking to also ensure that no updates are occurring while your query is running, but few applications need to be quite *that* current. Still, you could do so if you like.

To load-balance incoming requests across your shards, you'll probably want the shard manager to install a small routing table into the data center load balancing subsystem, directing it to send requests from client C to any of nodes {B,C,A'}, etc. These load balancers are quite elaborate and support high update rates. Thus Isis^2 is a good match with cloud-scale data sharding. You'll even get virtual synchrony (consistency) for the sharded data.

As explained in full detail later, the Isis^2 key-value store can be used in the manner that we've just outlined. The main difference is in the implementation: the key-value scheme doesn't create a true process group for each shard, and instead implements shards using an algorithm layered over the single encompassing larger group. This comes much closer both in spirit and in the details of how it operates to what Amazon does in its Dynamo key-value store, or what MIT's Chord DHT or the MSR/Rice Pastry system supports. Performance will differ (multicasts in a group are generally faster, but group management overheads are potentially higher). The Isis^2 DHT is fairly fancy: you can store data in memory, or can supply a "storage method" of your own, in which case you could store huge objects that need to live in external files. All of this is automatically managed for you by the platform. The DHT is also integrated with the .NET Language Integrated Query (LINQ) language features, and with the Isis^2 aggregation mechanisms.

We recommend using true groups if the amount of data to be stored will be large, and using the built-in DHT to maintain key-value pairs, but store the data (the values) in files if the actual objects are larger than about a half-megabyte each, employing the file name or a URL as the value. In our own research, the asymptotic performance of these kinds of solutions is emerging as an interesting question: we're actually still trying to understand which approaches work best, what sorts of overheads each approach incurs, and how to optimize the system itself in support of whatever scheme seems to be the most parsimonious in all of these different dimensions. The problem is fascinating… and we mention it just to help you understand that Isis^2 doesn't have all the answers, yet! It answers some questions, but in doing so, it also poses others.

More Fundamental Concepts

The central ideas in Isis^2 are thus multicasting to replicate requests (our Query API) or data (Send), groups and group views, and an event-oriented model that reports these kinds of events as upcalls to methods you register after creating the group and before joining it. Any group member can also implement a locally-responsive kind of query, performing an operation using its local replica of the group state and replying to the user (hence, the Query option is really used only for a kind of multi-RPC in which a request is multicast and then some number, or all, members of the group respond. If a request can be handled locally, there is no need to use Query).

At the end of this document we provide a table showing each of these APIs and the variants available to you as a developer. We also develop a fairly detailed example, in which we implement a token-based locking algorithm using the Isis multicast primitive to request and pass the tokens that represent locks.

Isis^2 users who are familiar with systems based on protocols like Paxos may be puzzled that Isis^2 doesn't need to use quorum[3] operations for updates or reads. The key insight here is that when Paxos obtains a quorum, it does so to overcome failures: because Paxos wants to tolerate *f* out of *n* failures without changing group membership, it allows an update to proceed even if only *n-f* group members receive it. Naturally, this means that to when working with quorum systems, even if you only want to read data, *f+1* replicas need to be queried.

In Isis^2, we never "give up" on a replica in this way: a multicast that will update a replica tries and tries and tries until the replica has acknowledged the update. The only situation in which Isis gives up on updating a replica is when a timeout triggers the membership protocol, which reconfigures the group and in so-doing, defines a new group view in which *n* may have a different value and certainly will have a different meaning: it will now relate to the new view, and the faulty member will have been dropped from the system. Indeed, if somehow it was partitioned away and tries to rejoin, it will throw an "I have received a poison pill" exception and be forced to restart before it is permitted to rejoin the main system. This philosophy basically keeps the main data center running even if we need to kill a few nodes or even a rack of nodes to do so.

But since we never update a subset of live nodes, we never could find ourselves reading a replica that somehow is out of date. In effect, we run with f=0 because rather than updating "during" a

[3] For those unfamiliar with the term, a *quorum* operation would be a read or write that touches multiple items within a replica set as a way to deal with failures. In a typical quorum scheme we have a group of size N members, and we define a quorum for writes, Q_w, such that any two writes will definitely overlap at one or more members. We also define a read quorum, Q_r, such that any read will overlap with any prior write. For example we might set Q_w=N-1, Q_r=2, and require that N>2. Now, by tracking versions, we can figure out the current value of any data within the group by reconstructing the history of updates. But notice that any single replica might be missing some updates. Also, notice that an update necessarily requires a two phase commit: when we apply the update, we won't know whether Q_w members were able to perform it successfully until after they report success to the leader. Thus, quorum schemes can be made fault-tolerant (indeed, we don't even need to drop members from our group when they fail), but bring a number of costs. In particular, nothing is local – even a quorum read needs to touch two or more copies, and writes have become multiphase commit operations. For those familiar with quorums, Isis^2 may seem like magic: replication without these costs! And as we'll see, this really is feasible. We've simply shifted costs by changing group membership on failure rather than keeping the group "static" even as some members fail or recover. Perhaps we should also mention that Paxos is a famous quorum-based protocol developed by Lamport, widely used in cloud settings.

failure, we’ll instead reconfigure the group, dropping the failure member. Thus, we don’t need quorum reads, and your applications can be locally responsive: running entirely on local data, and perhaps even entirely without locks. After all, most data has a primary owner and if all updates occur as multicasts from that owner, locks just aren’t necessary. You’ll end up with asynchronous updates that stream from the owner to the replicas, but every node in the group can still perform local operations, reading their local versions of the group state, with confidence that the data is either correct and current or, at worst, slightly stale. This is very much in line with the popular CAP and BASE approaches to cloud computing.

Let's Get Going!

At this point, you already know the basics. In the remainder of this tutorial we'll walk through details of the precise Isis^2 API and then show some examples of how everything comes together in some simple applications.

As mentioned at the outset, the current version of Isis^2 is limited to C# users under .NET, with access feasible from other languages (.NET supports 40 of them) but no documentation or support yet for doing so. Visual Basic or Visual C++ are probably next on the list, and then we'll do a port to Linux aimed at the Linux C++ and Java crowd. So all will come with time, but for now, you may just need to learn C#. Isis^2 has been tested under .NET 4.0 in Visual C# 2010, and on Microsoft Azure.

Threads

Isis^2 is *multithreaded* and this can be a *very* important thing to understand. In fact your code will be complete chaos if you don't master the C# threading model and learn to work with it. For example, if your code blocks for some reason while Isis^2 has issued an upcall to it, Isis^2 won't deliver additional events in the same group until your code is finished with whatever it was waiting for. This includes issuing a Query using the Isis APIs (since a Query waits for replies). Obviously, this makes it easy for you to cause a deadlock (at least in a single group). Thus you need to have a good mental picture of the threading model we use in order to avoid costly mistakes.

Things you need to know about this use of threads: Isis^2 has a number of its own threads, but in theory, they just do their thing in the background. But it also has one message delivery thread per group. These can be called concurrently with your code, and you need to protect any variables shared by your threads and by the handlers you register against read/write conflicts.

What about threads you create? Isis^2 assumes that you'll start the system from the "main" thread in your application. It monitors the thread that called IsisSystem.Start(), and if it sees that this main thread has terminated, then the library shuts itself down; this way your program won't linger in the background after shutting down. Of course this means that your main thread really needs to keep running. For some applications (e.g. GUI applications), that won't be a problem: the main thread will automatically be watching for buttons to be pushed and other GUI events. But some applications are designed to run as services, and for these it may not be so clear how to keep the main thread busy.

If you aren't sure what the main thread needs to be doing, call IsisSystem.WaitForever(). This method just sits within Isis^2, and it won't return unless something causes the library to shut down. To shut your Isis application down, you can call IsisSystem.Shutdown(). One cautionary comment: by default, this system call notifies other group members that your application is terminating and then calls the .NET "Environment.Exit()" system call, which is a fairly blunt-edged way to halt an application. Isis has code to shut down more gracefully, but we've had issues getting it to behave properly under all possible conditions (Isis^2 has many threads and many locks, and to get threads to exit gracefully we need to be sure that all of those locks are broken in the proper way). To enable this behavior, set the Boolean flag Isis.ISIS_GRACEFULSHUTDOWN to true. If you use this option successfully, when IsisSystem.Shutdown() returns control to the caller, the Isis subsystem on this node will have terminated and cannot be restarted, but your application could continue executing.

You'll probably create your own user-level threads if you become a serious Isis^2 user. If so, be aware that on .NET, an application remains "alive" until all its threads exit. Thus if Isis^2 shuts down or throws an exception, your threads might linger. We recommend that any user-level threads check the boolean flag `IsisSystem.IsisActive`, and exit if the flag becomes false. We will also throw exceptions such as the `IsisShutdown` exception if we know about your thread, but of course if your code is blocking outside of Isis^2, we have no way to do so. We suggest that you give your threads names (`Thread.CurrentThread.Name = "some string"`) and in one or two situations, discussed below, you may also need to elevate the runtime priority of a thread.

Beware of locks that you might be holding when you call one of the Isis^2 methods, especially the Send operations and the Query ones. As noted, Isis^2 could deadlock if you somehow wait for a lock while the system is doing an upcall into your code.

Important information about the way Isis^2 does locking: **Sorry about the bold font but this stuff is important. When you call into Isis^2, your threads acquire a form of mutual exclusion lock on the group associated with your Send and Query operations. Other calls to those operations will wait until the call holding the "entry to the Group" lock completes. Moreover, we acquire that same lock when doing an upcall to your code: we grab the lock, then call you. This implies that if your request handler blocks (for example by calling Monitor.Wait), *no additional messages or queries can be delivered to that group member in the same group until it wakes up!* Moreover, a *thread sending multicasts can block* because in Isis^2, multicast sends are subject to flow control. Thus, if you plan to send a burst of multicasts in response to a received view change or incoming multicast, it would be wise to fork off a new thread in which those sends will occur.**

Isis^2 uses two threads to deliver incoming messages and process-group views to your application. One is employed for point-to-point messages; these are delivered one by one, in the order they were sent. The second thread delivers both multicast messages and view changes, one by one, in the order Isis^2 computed based on the type of request and the ordering rules that it employed. If these threads block for any reason, Isis^2 will save new incoming messages and views on an internal queue for delivery once the blocking condition ends. If a delivery thread blocks for a long time, an exception will eventually be triggered.

What happens if you manage to trigger a deadlock by accident? Normally, this form of error will cause the system to eventually "kill" the node of your application in which the lock was held, so if you see crashes in which the system throws "poison pill" exceptions, deadlock could be the cause. You may think that Isis^2 was malfunctioning, but often the explanation is simply that your application created an impossible situation: we couldn't deliver incoming messages because your thread was holding the delivery lock. So they piled up, eventually creating back-pressure in the senders, which do wait for a while, but eventually kill off the receiver that seems to have gotten stuck. We think of this as a way of protecting the system as a whole against misbehavior by a single process.

The locking policy we use represents an important (and yes, annoying) limitation, and it is one you need to be aware of. Short locks, to protect shared variables using critical sections that don't block by calling Monitor.Wait, are not a problem. More complex synchronization may require that you create additional threads of your own: since you can't block while in the Isis^2 callback thread without causing the whole application to deadlock, you may find it necessary, in some situation, to create a new thread to take some action: that thread can safely block. New threads (even those created with inline code right in a request handler) are born without any locks at all. For example, here's a sample of code that uses the same "inline method" approach seen earlier. It creates, names and starts a thread that prints "Hello world" and then exits:

```
            Thread t = new Thread(delegate()
            {
                Console.WriteLine("Hello world!  I'm a new thread.");
            });
            t.Name = "An example thread";
            t.Start();
```

Thus, here, the inline method (the code highlighted in yellow) defines the logic that the new thread will run (a single line that prints "Hello world…"). It will run sometime after t.Start() is called but we can't predict how *long* the delay will be before that happens. Normally, it should occur within a few milliseconds. We'll use this form of callback all through Isis^2 because it allows the inner code block to access variables defined in the outer code block, with semantics very similar to those of procedure calls that pass variables to some method they call. But you can also write the method separately and then call

```
            Thread t = new Thread(myMethod);
            t.Name = "An example thread";
            t.Start();
```

This second example is probably more natural looking, at first glance: myMethod would simply be some method callable from this context, and by passing its name into the Thread constructor, it can do a callback later when the thread is running. Use whichever notation you prefer.

As mentioned, even though we didn't do so in our example, the code of the in-line version of the thread could have accessed variables from the scope in which it was created. Think of these as having the same semantics as code that might have executed at the same place in your program. This simple rule of thumb has several implications. First, read/write shared variables would need to protected using lock() or other synchronization mechanisms; otherwise concurrency conflicts (race conditions) could cause the values you read from them to be incorrect (in particular, C# will often cache values it loads from unprotected memory locations, hence without locking, your code might see "stale" values. Also, notice that we can't predict precisely when the new thread will execute: sometime after the Start() operation, but with no guarantees of exactly when. At any rate, this new thread could certainly use a Monitor to wait for something to happen, and it can call the Isis^2 primitives. Because it runs "separately" from the main request handler, however, it can't call Reply(). In Isis^2, the handler invoked to process a Query must reply to the query; if it neglects to do so, the system will automatically send a NullReply().

A second issue that arises with this sort of in-line code involves access to variables such as loop indices that change over time. Suppose we created a series of threads this way:

```
      for(int n = 0; n < 10; n++)
      {
            Thread t = new Thread(delegate()
            {
                Console.WriteLine("Hello world!  I'm thread “ + n + “.");
            });
            t.Name = "Example thread " + n;
            t.Start();
        }
```

As you can see, the intent is to create ten threads with names such as “Example thread 4”, each of which will print its version of the variable n. But this code is incorrect, because we can’t know when the new threads will run. They will access the variable n but the i'th thread might “see” a value of n anywhere in the range from i to 10, inclusive! Thus we could see two lines that both print “Example thread 4”, no line “Example thread 2”, and might even see thread 5 print “Example thread 10.”

In contrast, this almost identical code fragment works as the user probably intended:

```
for(int outer_n = 0; outer_n < 10; outer_n++)
{
      int n = outer_n;
      Thread t = new Thread(delegate()
      {
          Console.WriteLine("Hello world!  I'm thread “ + n + “.");
      });
      t.Name = "Example thread " + n;
      t.Start();
  }
```

The very small change is correct because each iteration of the code block creates a new local variable n, and the C# compiler will need to keep that variable “alive” for use by the inner code block associated with the delegate. Thus each thread will have its own instance of n, initialized by the int declaration and then never changed again. Understanding this point should enable you to write correct, thread-safe code in the in-line style we use thoughout the remainder of this document.

In some situations, you may find it useful to “hand off” the role of processing a Query from the callback thread (which really shouldn’t run for long) to some other thread that you create. If you do this, inform Isis that the new thread will reply to the Query, this way:

```
Thread t = new Thread(delegate()
{
    Console.WriteLine("This thread will send the reply”);
});
t.Name = "A worker thread for a query";
SetReplyThread(t);
t.Priority = ThreadPriority.AboveNormal;
t.Start();
```

The call to SetReplyThread() is only done from inside a Query handler method, and must occur after you create the thread that will send the reply, and before you call t.Start(). Naming the thread, as shown above, is useful to assist in debugging. As shown above, we’ve found in our own use of this technique that elevating the priority of the worker thread (the line `t.Priority = ThreadPriority.AboveNormal;)` can improve response time. It is not a good idea to use ThreadPriority.Highest; doing so can starve the Isis^2 system of needed execution cycles.

Just the same, this is a good time to note that our use of threads isn't just to improve for performance; indeed, Isis^2 doesn't use all that much execution time in any case and for non compute-bound code, threading won't give much speedup. We need threads to avoid deadlocks in situations where one process needs to wait for something that might happen locally or might involve actions by remote processes. If you want to achieve high performance in a .NET application (or any managed framework, like Java), use multiple heavyweight processes, one or more per hardware core, not multiple threads. Thread based concurrency gives little speedup (if any) on multicore machines because of lock contention associated with memory allocation, deallocation, and garbage collection and until this issue is resolved (if it ever is), the path to big speedups is just to run more programs, not to add threading to your existing programs.

Note: Isis^2 groups work well even on a *single* machine with multiple cores and even with multiple processes running on that one machine. We use these configurations in our own testing and they work well. For example, in debugging, we've often run as many as 20 copies of a test application on a normal dual-core laptop without problems, and for certain styles of applications have run 50 on one machine successfully. This is very helpful when developing scalable applications and we highly recommend that you experiment this way before renting lots of cores on Azure or Amazon's EC2!

Callbacks

Many Isis^2 notifications are delivered via callback to methods you declare either inline or as C# procedures that you register with Isis^2. It is important to realize that the code in these callback handlers could run long after you declared them, and in a different thread. Think of a callback, even an inline one, as a method that will be called with whatever arguments you specify, as well as with additional arguments corresponding to the variables from the enclosing scope that your code uses. Those values are effectively stored as a kind of snapshot when you register the handler, almost as if the handler was partially invoked at the instant it was declared (to capture values of those in-scope variables), and then the invocation completed much later, when the event itself (perhaps, a arriving message or a new View) triggered the callback action.

Address Objects

An Isis^2 Address is a opaque endpoint representation for a process using the library. It encodes the IPv4 address of the process, the pid on that machine, and two port numbers. A variant form of Address is used as an internal endpoint to identify an Isis^2 Group without needing to pass its name around (group names can be strings of arbitrary length; applications that use many groups might want to use a file-name like syntax for them. But this also makes it awkward to pass group names around since they can be long). In this variant form, the Address contains a logical IP multicast address for the group. As noted above, Dr. Multicast is employed to map these logical IP multicast addresses to either a physical (and perhaps, shared) address, or if no IP multicast address is available, to UDP endpoints for the members. Note: IPv6 support is planned for the future.

How do you "learn" the address of a process? When the process is first launched it calls a function IsisSystem.Start(). (After doing so it may just use Isis^2 forever, or it could eventually call

IsisSystem.Shutdown() to disconnect from Isis^2; if so, this version won't allow a later reconnect). When IsisSystem.Start() returns, a call to IsisSystem.GetMyAddress() will return the address that will be used for this process during this run.

But of course with Isis^2, your focus will be on applications composed of multiple processes, each with its own address, that come together to form groups (see the next section). Very often you'll have some form of leader or master process that coordinates the initialization; how can it learn the addresses of its "workers"?

One possibility is to launch the worker processes, have them each learn their own addresses, and then pass these to the master. You could do this, for example, by converting the Address to a byte[] vector (e.g. byte[] myAddressAsBytes = IsisSystem.GetMyAddress().toBArray()). This byte string can then be written into a binary file, sent over a TCP link, etc. Another option is to have your new process join a process group that the master process would monitor; the downside of doing so is that if you launch 1000 processes all at once, the group membership will take a while to settle down (in contrast, having the master process create a group with an initial membership of 1000 processes would be very fast). The fastest solution of all is to issue a "MultiJoin()" request that lists multiple groups and multiple addresses, and just initialize everything in one action. We'll say more about this below; it isn't hard but requires special incantations.

Note: The Isis^2 address of a group will be constant over incarnations of the group, hence it is safe to treat the group address as persistent data, should you need to do so. The addresses of its members, in contrast, include process id numbers and can change from run to run and can be reused if your operating system happens to reuse a process ID after some long delay. Thus, it is not safe to consider a process address as persistent data.

Messages and Marshalling

Internally, Isis^2 makes extensive use of a data structure called an Isis "Message" (class Msg). A message has a header listing the sender (or creator) process address, the destination address (a group address, or a member from the active view of that group), a view id for when the message was sent, a message id, and a "payload" consisting of a byte[] vector.

The payload of a message encodes some set of objects and their types. Each type must be known to Isis^2 either because it is a primitive type like int or double, or because the type is an instance of a class registered via Msg.registerType(), as discussed below (If you are working with a more complex type, you may want to drop a note to the Cornell development team to see how best to "tell" Isis^2 about it). Isis^2 knows about a few fancier types, including Arrays (1-dimensional for most types, 2-dimensional for a few), .NET List<T>, KeyValue<KT,VT> and some others.

You can convert an array of type object[] into a byte[] encoding by calling Msg.toBArray(). The resulting vector of bytes can be converted to an array of objects by calling Msg.BArrayToObjects(). If you know the type of the expected objects, an overload of Msg.BArrayToObjects() allows you to specify them and will throw an exception if the byte[] vector encodes the wrong number or wrong types of objects. You can create a new Msg object by calling the Msg constructor with a list of objects that will form the payload of the message.

One small caveat applies: if your new message will contain a single byte[] vector as its payload, ambiguity arises: Isis^2 defines a new Msg(byte[]) method which is used to demarshall a message from byte[] outform into a Msg() object. Accordingly, if you wish to create a message that will contain a payload consisting of a byte[] of your own, cast your byte[] to type object, as follows: Msg myMsg = new Msg((object)myByteVec); This will ensure that C# selects the appropriate constructor. If you fail to do this, you will probably get an exception from Isis^2, complaining that it expected a vector of 6 objects of various types, but found something else: the system thought you were trying to demarshall your byte[] vector into a Msg.

There are situations in which you may find it useful to access the Msg data marshalling routines directly. For example, in an appendix to this manual we discuss the challenges of running Isis^2 with very large groups. In such cases, if every member joins willy-nilly, the load on the Oracle gets very high and the whole system can collapse. Instead, multi-join and multi-leave APIs are used to manage these events, adding batches of members all at once, etc.

To carry that sequence out, as new members are launched, they need to pass their Isis^2 Address to the manager, so that it can add them to the system and to any groups they might want to join. One does this by starting the worker applications using a special Isis^2 API (described later), then marshalling the Address each worker is assigned into a byte[] using IsisSystem.GetMyAddress().ToBArray(), and then passing the resulting byte[] vectors to the master process, in any way you like. It can then turn them back into addresses using new Address(byte[] stuff), and "viola": all ready to call the MultiJoin methods. Another way to do this is to call Msg.ToBarray(obj1, obj2, ….) to turn a set of objects into byte vectors, and then call Msg.BArrayToObjects(byte[] stuff), to turn the byte vector back into an array of objects, each having the same type as the corresponding object when the byte[] vector was created.

Groups

An Isis^2 Group is a local object representing a kind of handle on a distributed group containing the members. Each group is typically associated with some user-defined class that uses the group, and that class will often have a state that is partly replicated and partly local to each member. The local state, in turn, may be defined over some set of user-defined classes.

Accordingly, when using a group, there are several steps. First, the user creates an inactive group handle and registers any data types or handlers that will be used. Next, the group handle is activated using a group Join or Create operation. And finally, now that the process is a full fledged member, there are operations to multicast into the group, query it, monitor view changes, etc.

Below is a small Isis^2 program illustrating all of these features, although the code is nonsense. So what's going on in this example? The logic is trivial but the syntax may be new to you. First, we launch Isis^2 itself; this is always the first action in any Isis^2 application. Next, we create our group, giving it the name "some name". In practice this would more often be a file name in the global file system of the network or data center, but any string will do. Every member uses the same name, of course.

Next, we associate a handler with the LOOKUP request id: this particular handler is an in-line method defined using a C# notation that does just what it looks like it does (technically, C# creates an anonymous class, but the details don't matter). The handler will run later, when LOOKUP requests are received. Notice that it has access to variables from the context in which it was declared. Think of these as having exactly the semantics of arguments to a method: myGroup, for example, was "saved" at the time the LOOKUP handler was declared, and can be accessed from within the handler, as we see in this example. You can also specify that the handler expects various arguments from the message that will trigger the call to it. In our example, this particular handler expects a string argument.

Next, we activate the group by calling myGroup.Join(). And then we fire off a Query. In this particular example, the Query expects replies from ALL group members (as defined by the View at the time the multicast is sent; Isis^2 will figure out how many are really running when that happens). The Query timeout specifies that we should wait for 1 second for members to reply (1000ms), and indicates that a slow member should be assumed to have crashed (TO_FAILURE). The request is LOOKUP, and the argument is a string, "John Smith". That's the end of the list of arguments for Query method (hence the EOLMarker object).   By the way, Group.EOL is a pre-allocated object of this type, and is safe to use if you prefer not to create one of your own. The EOL marker doesn't need to be distinct; Isis^2 only cares about its type.

We needed to provide a place to put the replies. Keep in mind that we don't know how many members this group actually has, so there will be *n* replies, one from each. The members reply by sending an int (refer back to the myGroup.Reply() call in the handler), and in fact each sends its own rank. Isis^2 collects these into a vector, which it resizes to match the value of *n*. So we sent "in" a 0-length array (to avoid an uninitialized variable complaint from C#) and got back an array of length n, which we print. Of course C# isn't very smart about printing arrays, so we use a little loop to iterate over the elements. If nothing crashed, you'll get nr equal to the number of members and will find the numbers 0..nr-1 in the array, in whatever order the replies came back.

Often you'll supply multiple lists, one for each of a set of replies (e.g. the code might call g.Reply(x, y) and your caller will have a list for the "x" objects, and a list for the "y" objects. We guarantee that the i'th item in the list of "x" objects came from the same sender who sent the i'th item in the "y" list. This is often helpful (for example, if x[i] is the rank of the replying process, y[i] is the rank-i contribution to the result.

Isis^2 checks to make sure that the replies match the lists provided to collect them. Thus if "x" is of type foo, and the corresponding list is of type bar, or if the number of replies doesn't match the number expected, an exception will be thrown.   However, there is one exception to the rule: you can ignore replies by omitting the EOL marker and omitting the lists. In this case, the Query is performed and Isis^2 waits for all the replies, but it doesn't keep them and won't check for a match of number or type, since of course the caller is ignoring them in any case. The effect is a bit like a void call to a function that returns a value.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Isis;

namespace ConsoleApplication3
{
  // Type signatures.  C# really ought to infer these from the context
  delegate void myLhandler(string who);

  class Program
  {
    public static Timeout myTO = new Timeout(1000, Timeout.TO_FAILURE);
    public static EOLMarker myEOL = new EOLMarker();
    int LOOKUP = 0;
    static void Main(string[] args)
    {
      IsisSystem.Start();
      // First create a group that includes just a single handler
      Group myGroup = new Group("some name");
      myGroup.Handlers[LOOKUP] += (myLhandler)delegate(string who)
      {
          Console.WriteLine("My LOOKUP handler was asked to look up {0}", who);
          myGroup.Reply(myGroup.GetView().GetMyRank());
      };
      myGroup.Join();

      List<int> ranksList = new List<int>();
      int nr = myGroup.Query(Group.ALL, myTO, LOOKUP, "John Smith",  myEOL,  ranksList);
      string reps = "";
      foreach (int rep in ranksList)
           reps += " " + rep;
      Console.WriteLine("Got {0} replies and they are {1}", nr, reps);
      IsisSystem.Shutdown();
    }
  }
}
```

So this illustrates the one-to-all and all-to-one pattern of our first figure. In our example, we're use a timeout structure to indicate that the lookup should wait no longer than 1 second (1000ms) for replies, and that a node that doesn't reply within this limit should be treated as having crashed and dropped by the system. Had we omitted the timeout argument, a default would have been used: 15 seconds wait, followed by an AbortReply that would cause the Query to throw an IsisAbortReplyException. Next, let's register a multicast handler for UPDATES. We'll do this in a way that illustrates polymorphism:

```
    delegate void stringArg(string who, int val);
    delegate void intArgs(int id, int val);

    class Program
    {
        static void Main(string[] args)
        {
            IsisSystem.Start();
            int UPDATE = 1;
            Group myGroup = new Group("some name");
            myGroup.Handlers[UPDATE] +=  (stringArg)delegate(string name, int val)
            {
                        Console.WriteLine("Update {0}, new value {1}", name, val);
            };

            myGroup.Handlers[UPDATE] += (intArgs)delegate(int id, int val)
            {
                        Console.WriteLine("Id-based update {0}, new value {1}", id, val);
            };
            myGroup.Join();

            // Now send some multicasts
            for(int n = 0; n < 10; n++)
                        myGroup.Send(UPDATE, "User number " + n, n*100);
            for(int n = 10; n < 20; n++)
                        myGroup.Send(UPDATE, n, n*100);
            Thread.Sleep(10000);
            IsisSystem.Shutdown();
        }
    }
}
```

If you run this program, all the program instances that are concurrently active will receive twenty multicasts. The first ten will trigger upcalls from Isis^2 into the first of the handlers and the second ten, which have an (int,int) type signature, will trigger upcalls to the second handler. These upcalls run in a separate thread associated with Isis^2 itself and are concurrent with your Main thread, and we need to give the system enough time to send the multicasts and deliver them, hence the 10-second sleep before the program disconnects from Isis^2 and exits.

In general, the rule is that for any given incoming message, Isis^2 invokes all the handlers that have matching request id’s and type signatures.

Flavors of Query and Multicast

Isis^2 actually offers several versions of the Query anD Send interfaces: Send, OrderedSend and SafeSend, and UnOrderedQuery, Query, OrderedQuery and SafeQuery. The Queries use the Sends, so we can just explain the basics once. Note: there are also completely "unordered" variants (UnorderedSend, UnorderedQuery) but we don't recommend using them.

A Send is basically an IP multicast, synchronized with respect to view changes and made reliable but otherwise sent (asynchronously) when you invoke Send, and delivered as soon as the members of the group receive it. The speed is as fast as the data center can do IP multicasts, and the latencies can be very small.

But Send only promises FIFO ordering relative to the sender. In our example above, one sender sent 10 messages, then 10 more. They'll be delivered in the order we sent them. But if two or more senders compete, messages *from different senders* can interleave in any which way.

Moreover, if the sender and some receiver crash, we can get a situation in which the message was delivered and caused some action (as Leslie Lamport likes to say, "the rocket was launched") but then completely forgotten: nobody else in the group will see that multicast. This can only occur during a very brief window of vunerability: after the multicast Send occurs, it would need to be delivered to some members, but some of the messages would need to get dropped by the network, and then all the members that did receive the multicast, plus the sender, all need to crash. This will cause the evidence that the multicast was ever in the system to be erased, hence it will be forgotten. In our lab experiments, we've only produced this behavior by partitioning the network in such a way that the sender and those initial recipients are isolated by the failure, but with some additional group members still running in the main system. As an example, we see a possible instance of this pattern in Figure 1 on the lower right: the single stray message sent from p to q after the remainder of the system believes they have failed. But obviously, such a sequence of events isn't at all likely.

As noted earlier, we recommend that Send be used for soft state. In this situation, and indeed for many other purposes, the small risk of amnesia we've described poses no risk: in effect one uses Send in situations where updates don't need to guarantee stronger durability.

If you do use Send, you'll gain better performance and better scalability. One way to have your cake and eat it too is to use Send but then call the Isis^2 Flush primitive prior to communicating with an external user or writing data to disk: this delays your code very briefly (a few tens of milliseconds) until the pending Sends (those active in the current group) are safely delivered. Now, your code is at no risk at all: that window of vunerability has closed.

The OrderedSend is like SafeSend, but adds a guarantee that even if multiple senders are sending concurrently, everyone gets all the multicasts in the identical order. This can be very helpful if you are writing order-sensitive update code. But it slows things down quite a lot in some conditions, because we need to wait to learn the ordering from the rank-zero group member. That member has an advantage: its OrderedSends are automatically done as normal cheap Sends. Again, no guarantee of durability.

Last and strongest are the SafeSends. These are ordered and also durable: if anyone receives such a message, everyone will. But to ensure this Isis^2 has to delay until it has confirmation from a majority of the group members, which can be slow. So don't use SafeSend unless you are sure that the kind of durability it offers is important to you. You'll be switching from speed of light to speed of sound, and they are very different numbers!

Finally, we should note that Isis^2 offers several ways to obtain the replies in a Query. Earlier, we saw an API that returns *n* replies of types t_1, t_2… as a series of Lists<t_i>, each *n* elements long, the first containing elements of type t1[], the second of type t2[], and so forth. The user creates an empty List<t_i> variable, and Isis^2 uses List.Add to append elements to it as replies are received.

A second option is to receive the replies in an List<byte[]> array: a set of *n* byte vectors. Here, response[i] is a byte string marshalling the i'th reply that Isis^2 received. One calls the Msg.BArrayToObjects() method to convert such a byte string to a vector of typed objects. Yet a third option is to call Msg.InvokeFromBArrays(). This will code a method, or an in-line delegate declared much as we did earlier for our handler methods, passing the replies in as arguments. One ends up with code that looks like this:

```
List<byte[]> ba = myGroup.QueryToBA(Group.ALL, myTO, LOOKUP, "John Smith");
Msg.InvokeFromBArrays(ba, (intVecArgs)delegate(int[] ids, int[] val)
{
  Console.WriteLine("Got {0} replies to my query.", ids.Length);
  for(int i = 0;  i < ids.Length; i++)
          Console.WriteLine("   … Reply {0} has id={1}, val={2}", i, ids[i], val[i]);
}
```

The inner block of code will receive vectors of length *n* if *n* replies were received. Be aware that *n* could be zero, if the group failed or if all members sent NullReply responses!

Options For Collecting Replies

Variations on the basic Query and P2PQuery APIs allow you to collect replies in any of three ways. Above, the replies are de-marshalled (unpacked) into List<T> objects you provide, where T is the type you expected back. You can also ask for replies as byte arrays: byte[] for the peer-to-peer calls, and List<byte[]> for the group Queries. And finally, Isis^2 can do a callback to a delegate of yours, which can be declared inline (our favorite style) or as a method.

Recall that if there are multiple reply lists, for the "x" objects and the "y" objects, then Isis^2 maintains alignment: reply x[i] will be from the same source as reply y[i]. Also, keep in mind that you can completely ignore the replies, but if you provide a list to capture any of them, the number of lists and their types must match the number of objects in the replies and their types.

Request handlers

A request handler is declared in the same manner as a multicast message handler. On receipt of a query, Isis^2 will find the handler that matches the type signature of the Query message (there should be exactly one such match; if there are multiple matches, buggy behavior is likely). The handler will be invoked, and should compute the result and then reply via Reply(), NullReply() or AbortReply(). If no reply is sent, a NullReply() will automatically be performed. Note that if a thread is spawned by the handler, this basic policy still applies to the handler even though that thread might still be running. Moreover, *only the thread Isis^2 invoked to handle a request can reply to that request.*

As mentioned earlier, until a handler returns, other messages in the same group can't be delivered, because the handler holds an exclusive lock (see earlier discussion, about Threads). What this adds up to is the following: if a request will execute slowly over an extended period, Reply promptly when the request is first received. Send back some form of "in progress request ID" to the caller if you wish, but don't delay the Reply(). Later, you send back the result as a separate message, using P2PSend; if needed, identify the data using the request ID you produced during the first step.

Query calls can throw an `IsisAbortReplyException`. If this occurs, the reason that was specified in the call to AbortReply will be passed up as the exception contents. The Query caller will be interrupted. However, other nodes that received the query may still be executing: AbortReply won't impact them in any way, and the results they return, if any, will be sent to the Query caller but then silently ignored.

Flush

As mentioned earlier, Isis^2 offers a primitive you can call to pause execution until any unstable multicasts (that is, any Send or OrderedSend multicasts that haven't yet been fully acknowledged) and that are associated with a designated group have reached their destinations and been acknowledged. This operation is called a group Flush and there isn't much mystery to it. The length of the delay depends on how many unstable messages Isis^2 has in its outgoing buffers at the time of the call. It can delay for a long time if you've issued a high rate of asynchronous Send operations and they are piled up in the communication layer. It may return instantly if you haven't sent much recently. In our experiments with Flush, we generally saw delays of a few milliseconds and rarely saw delays that exceeded 20 milliseconds even in very large, heavily loaded groups.

Flush is not necessary in groups that only use SafeSend and SafeQuery.

Failure Sending and Reporting Application-Detected Failures

Isis^2 has a variety of failure-sensing mechanisms that run in an automated way. These will generally declare a crashed process as faulty within a few seconds. If the process wasn't really dead but was just very slow or temporarily unreachable, when it regains connectivity the system will send it a "poison pill" message, causing it to shut down.

Applications can augmented this behavior with logic to "sense" and report failures. Recall that a Query can timeout, declaring a process to have failed. Some time will elapse between when this occurs and when the new View is reported in which the failure becomes official. During that period, you can actually see that the member has failed by interrogating the current view, but not all members will be aware of the failure until the new view is actually reported. There is also an interface, g.HasFailed(Address who), with which any application can report that any other process in the current view has failed. Obviously, this feature must be used with care! It is only possible to report a failure for a current member of some group to which the caller also belongs.

Unreliable Messaging

Isis^2 supports an API aimed at developers of gossip protocols and other applications that require access to unreliable datagram communication options within a group. The g.RawSend(), g.RawP2PSend(), g.RawQuery(), g.RawP2PQuery() and g.RawReply() methods send messages that will not be acknowledged on receipt and will not be retransmitted in the event of message loss.

These methods thus offer a direct form of unreliable datagram communication, performed within the Isis^2 infrastructure.

One caution: even though raw messages are not directly acknowledged, Isis^2 will still send flow control and other overhead messages, hence applications using these features will not have full control over the lowest level communication traffic. Further, because encoded packets that are larger than ISIS_MAXPACKETLEN bytes must be fragmented, the risk of loss rises as a function of packet length. If a long packet is sent, fragmented, and then some fragments fail to arrive, space will be consumed on the receiver process until the system eventually realizes the message cannot be fully reconstructed and the partially reconstructed fragments are discarded. Thus, it is unwise to use the raw APIs to send large objects. Keep in mind that in addition to any data bytes the developer provides, Isis^2 adds a number of headers of its own and that a packet may have as much as 1000 bytes of overhead in extreme cases, over and above the encoded size of the data you place within it.

A second caution runs in what may seem to be exactly the opposite direction. RawSend, although subject to some forms of flow control, is actually very capable of overrunning the receivers by sending bursts of packets at very high rates that might overwhelm the capacity of their UDP receive sockets, which have a backlog set to 128 times the maximum packet size. Thus if a receiver lags the sender even briefly it can drop packets, which will not be retransmitted. With larger packets that fragment into pieces this becomes a serious problem, because if a single fragment is lost, the larger packet will never be fully reassembled and within a second or so will be discarded. Thus it is relatively easy for RawSend to find itself in a state in which no packets get through at all.

To avoid this issue, one suggestion is to limit use of RawSend to smaller packets that won't need to br fragmented. For example, with a typical UDP socket MTU of 8k, you might consider limiting yourself to about 5k of application data. A second suggestion is to consider using the rarely needed parameter ISIS_RATELIM, which specifies a maximum packet sending rate in UDP sends per second. For example, ISIS_RATELIM=1000 would limit the system to sending 1000 packets per second. If the rate ever rises above ISIS_RATELIM, small delays are initiated by the sender to slow the rate down. In our own experiments with RawSend, we found that setting ISIS_RATELIM=2000 worked well and avoided most problems, and in an otherwise idle cluster, we were able to run with ISIS_RATELIM=2500, although we saw issues on that same cluster when we put significant background loads on the nodes. Thus you may need to experiment in your own setting. The peak sustainable rates will depend very strongly on the hardware you use.

Flow Control

Isis^2 tries very hard to strike a balance between supporting high data rates and avoiding situations in which some nodes overwhelm other nodes with such huge torrents of data that the senders or receivers crash: the sender could crash if it gets way ahead of the receiver, in which case it would need to buffer vast amounts of data and eventually would run out of virtual memory, begin to page (thrash) and be killed off for poor responsiveness. Or a receiver could receive messages but not be able to keep up with processing them, bloat out by having more and more of them enqueued for delivery to the application, and eventually crash in a similar way.

Thus Isis^2 uses a variety of algorithms to estimate backlogs both on the sender side of a multicast series and on the receiver side, attempting to choke back a sender so that the rate of sending an the rates of receiving will be matched. We also offer you, the developer, a leaky-bucket rate control scheme that you can use to smooth out bursty sends, if you find that our scheme isn't working well in your particular setting.

Flow control in Isis^2 is best visualized much like flow control in TCP: the system will generally send a burst of messages at very high rates until a backlog of unacknowledged (also called *unstable*) messages begins to form. Then the system will choke back, waiting for acknowledgements, which flow back to the sender through a variety of channels: direct acks and nacks, stability information piggybacked on other messages, etc. Once the overload drains from the system, a new burst of sending can occur. Thus one sees a kind of sawtooth behavior, much as for the famous TCP window size or bandwidth graphs published in so many introductory networking textbooks.

Isis^2 should be able to deal in an automated manner with loads that originate mostly at one sender at a time, even within large groups, and should achieve high rates and low latencies for such cases. But if your application generates traffic from many senders in overlapping patterns, it isn't hard to confuse our flow control scheme. Generally, the error would involve being over permissive: we might allow messages to be sent without realizing that a remote receiver is falling behind. You'll see bursty behavior, then the receiver in question will probably seem to hang (if you watch closely you'll see it bloating as messages pour in so much faster than it can process them that it just holds on for dear life) and then finally, the sender will declare the slow receiver to have failed and will move on. This can take 30 seconds to a minute or so to play out, since our failure detection threshold is based on a 20-second timeout.

The choice of multicast primitive has a big impact on how well our flow control policies work. Right now, the basic FIFO Send, OrderedSend and SafeSend seem to do better than CausalSend: the former appear to work well in all patterns we've experimented with, while CausalSend has difficulty in larger groups with many senders and high data rates, where the causal ordering property is hard to enforce because the causal "out of order" condition arises frequently. In such cases, a backlog of undelivered (because out of causal order) messages can form, and these cause flow control to kick in more often. Thus the aggregated throughput can be reduced – but only if this pattern occurs in your application.

The issue illustrates a deeper challenge: with multicast at high rates in big groups, it isn't hard to "attack" a process in such a way that it will drop messages and, when that happens, you can easily get into a situation where catching up is just not possible. Use our leaky bucket rate controller to prevent such problems if you encounter them. For our part, we're working hard to improve our flow control methods, but the question is another one of those fascinating research topics. So don't expect dramatic progress overnight!

Tabular Summary of Multicast, Query and Reply Operations

The tables that follow summarize these operations and their variants. Each is associated with an active group instance (the notation g.Send()… is omitted to simplify the table) We'll start with the basics:

Operation name	When to use it
Send	Fast, FIFO ordered multicast. Reliable (recovers lost network packets) among processes that don't crash, but rare failure patterns could allow a Send to then be delivered at a small number of destinations, but then "forgotten," if the sender and all recipients fail but some other group members survive. This is the workhorse protocol in most Isis^2 applications because it maps directly to IP multicast. The virtual synchrony model ensures that *all members in some view* will receive the multicast, and if each member checks the view upon receipt of the incoming message (via g.GetView) all will see identical View contents. There is a limit on size of messages that can be sent using Isis^2, but the actual value of the limit will depend upon the maximum size of packet that Isis^2 is permitted to generate (see ISIS_MAXMSGLEN). With an 8KB maximum, the largest legal Isis^2 payload will be about 1.5MB. If the maximum is increased by a factor of X, the maximum will also increase by a factor of X. Our plan is to experiment with extremely large packet sizes in the future, but in this initial release, we've only tested with relatively modest sized messages.
OrderedSend	Like Send, but guarantees a total delivery ordering even if sent by some other group member: every message is delivered in the same order at every group member. Reliable but not durable. Respects the FIFO order used by the sender, but extends it into a total order across all concurrent sends. OrderedSend is slower than Send except in one special case: the protocol optimizes itself in situations where *the sender is the rank-zero member of the group* and *no other member has sent any ordered multicasts during the current view*. It will run faster while this condition holds. As soon as some other member does use OrderedSend, the optimization shuts itself off, and the rank-zero member's multicasts then have the same cost as multicasts from any other member, until the next membership view change event.
RawSend	Like Send, but with no guarantees of reliability.
Query	A Send followed by a wait for replies. If a member sends a NullReply or crashes before replying, the reply count won't include it. If it replies and then crashes, but the reply gets through, the data from that member will be included.
OrderedQuery	Just like Query but uses OrderedSend to send the request. This has the advantage of being totally ordered with respect to updates that were sent using OrderedSend. Thus the group members will be in identical states (same group view, same data) when they process the query.
RawQuery	Like Query, but with no guarantees of reliability. If the RawSend fails to reach some members, or their replies are sent with RawReply and fail to reach the sender, the RawQuery will timeout and apply the timeout action specified by the developer.

Flush or Flush(k)	Delays the sender to wait for pending multicasts to reach their destinations (Flush waits for pending multicasts to reach *all* destinations, while Flush(k) waits until pending multicasts have each reached at least *k* destinations). Once Flush has returned, any multicasts that were in progress at the time of the Flush will have been delivered to the application and hence are durable.

These already cover most of the scenarios encountered in normal use of Isis[2]. But the system supports a few more options for those who want to implement some of the more elegant, sophisticated group communication algorithms that one finds in the literature (which is extensive):

Operation name	When to use it
CausalSend	Like Send, but extends the FIFO delivery property of Send into a causal delivery property. To understand this property, you'll need to know about the notion of causal order, as defined by Lamport. We'll assume you do know about this definition (if not, don't use this primitive). In a nutshell, if CausalSend X is issued before CausalSend Y in some group, then X will be delivered before Y even if the senders are different. Causal order hinges on the definition of the term "before:" when does the system consider Y to have been sent after X? There are two basic rules. First, if some single sender sends X and later sends Y, Y needs to be delivered after X, just as in a FIFO ordering. The second rule is trickier: if X is received by a process that then sends Y, this means that the sender of Y knows about X. In this case, CausalSend would also deliver X before Y at all the other group members. Notice that a causal order is weaker than a total order: if X and Y are sent concurrently by different senders, and the receiver of X hasn't seen Y or vice versa, then no ordering is imposed and they could be delivered in different orders at different receivers. With CausalSend, performance will be similar to that of Send (a tad slower in cases where Y arrives before X and yet that second ordering rule says that Y can't be delivered until after X has been). CausalSend will often be than OrderedSend (much faster than SafeSend). The ordering policy is very useful in conjunction with locking on shared replicated data, as explained in Ken Birman's textbook and research papers. But again, don't use this primitive if you don't understand the policy or why it can be useful: CausalSend really is *only* useful in algorithms that are proved correct for this particular ordering rule!
CausalQuery	Just like Query but uses CausalSend to send the request
SafeSend	A slower but very robust protocol. Delivery is reliable (overcomes packet loss) and totally ordered but also won't occur until a majority of receivers have logged the message. This is exactly equivalent to the Paxos protocol.
SafeQuery	Just like Query but uses SafeSend to send the request

The wide range of multicast and query options can be confusing, and one can make a case for omitting at least some of these. Leslie Lamport, in developing Paxos, ended up favoring just a single primitive (SafeSend). Yet that choice compromises performance and scalability to gain simplicity, and in cloud settings, performance and scalability are primary needs.

In fact, as noted earlier, many users limit themselves to using Send or OrderedSend, calling Flush prior to replying to "external" clients, and using Query or OrderedQuery for parallel queries in the group. These are easy to understand: Send (and Query, which runs over Send) are like a the TCP FIFO property, but now with one sender and multiple receivers. One uses OrderedSend or OrderedQuery if there might be multiple sources of multicasts in a single group, and yet we want them delivered in a single ordering, for example so that a sequence of updates will be applied in the same order at all replicas.

So why offer all these other options? There are really two kinds of reasons. The more important-sounding one is that the other options can be useful in implementing more sophisticated asynchronous algorithms. This is especially true for the CausalSend and CausalQuery. Yet we are quite aware that many (perhaps most) developers will never need to learn how they work or why they exist. Don't feel any sense of obligation to do so unless you are curious about the topic, or are worried about the risk of relatively obscure delivery ordering issues that might violate application correctness.

SafeSend takes us even further into a complicated issue, namely durability. As mentioned earlier, SafeSend is our name for the Paxos protocol; the key guarantee it provides is that if any group member delivers a message, every member will deliver that message, unless it crashes first, and in the same order too. This is a useful property when communicating to the outside world: if a recipient talks to an outside user, or updates a file or a database, it can be very helpful to know that all the other replicas, or users, will see the same thing. One difference between SafeSend and Paxos is that in most large-scale Paxos deployments, one actually uses two groups: a group of *acceptors* (think of these as processes that maintain a database, probably using a quorum architecture) and a second group of *learners* (think of these as being, for example, cache servers that monitor the database and keep read-only copies of popular items). But in Isis^2 a single group plays both roles.

More specifically, in Isis^2 one uses SafeSend within some group, specifying the number of members that should play the *acceptor* role. These will be the first Ω members in any view (you get to specify the value for Ω). The idea is that the acceptors will assure the durability of the multicast, but then it will be delivered via the standard upcall API in all group members: all members are *learners.* A second difference relates to how one uses SafeSend: very often, in Isis^2, an application is somehow wrapping a service, for example by maintaining one replica of a database or file or some other kind of application next to each group member. The multicasts contain inputs that are passed to these replicas on arrival.

Notice the inversion of layers: in standard Paxos deployments, the acceptors *are* the database. With Isis^2 applications, the group is more likely to be a communication front-end that *talks to* the database replicas. As we'll now see, this leads to some details one wouldn't normally discuss in a paper on Paxos. On the other hand, relatively few researchers have ever really worked with Paxos in situations where it functions as a multicast and talks to some form of external service.

The complications involve the handling of total group failures: cases where a message is in the system, and while it is being delivered, *all members of a group crash* (perhaps, Amazon West just had a data-center wide power outage). With SafeSend we run into an issue here: what if a message was delivered to some group members (but not all), and then the group crashes. Later it restarts. Should we replay that partially delivered message? Not worry about it? In fact Isis^2 lets you decide:

- You can call g.SafeSendThreshold to tell the system how many "copies" of a message are needed for durability. For many purposes, 3 seems to be best. But any value up to the full size of the group is permitted.

- Finally, you can provide a durability method of your own, implementing the IDurable API. With this approach, you gain a great deal of control over precisely how recovery will occur after a disruptive failure.

To understand these choices better, we recommend that you start with the basics: by appreciating the difference between OrderedSend and Send. With OrderedSend, you pay slightly more in terms of a slower delivery, but gain total order. In contrast, Send only guarantees FIFO ordering: if a single source sends multiple messages, they arrive in the sender order. But there are some cases where OrderedSend and Send will be identical. Think about an application in which all the updates for a given kind of data originate at the same process. Here, you should use Send: your code will be faster and the executions that result, identical!

CausalSend extends the idea of Send to cover execution threads that "cross" process boundaries (e.g. P sends X, which causes Q to send Y). Interestingly, CausalSend is basically as fast as Send, although it will delay out-of-order messages that Send would have delivered promptly. To appreciate the value of this, imagine a system in which the role of being the update source sometimes moves around, perhaps from P to Q. Can you see why Send might result in out-of-order updates here, but CausalSend never would?

Finally, look at SafeSend. When using this protocol, you pay quite a bit more, but in principle (if you set the various properties correctly), are protected against data loss in the event of even extreme failures, even if they occur just as the protocol is running. The basic guarantee here applies when the group doesn't experience a total failure, and is this: if *any* group member delivers a message, *every* non-failed member will do so, even if that first member crashed the instant it started to do the delivery upcall. To see how this is useful, consider an application that does something, like issuing cash from an ATM, and imagine that the only process that hears about some update is also the one that issues the cash, and the one that promptly crashes. Clearly, SafeSend is the better choice for sending those updates. With Send, money could be dispensed and yet there are patterns of failures that would erase the record.

The really challenging problem is to use SafeSend to keep duplicated copies of a database or file, and in which a restarting process, recovering from a crash, will need to reuse its local copy of the database, bringing it up to date by applying a "delta" of missing updates. You shouldn't have much trouble coming up with a scenario in which the weaker Send primitives could leave that database replica in a corrupted state. SafeSend offers a solution, but not a trivial one.

To appreciate the issue it helps to think about three distinct cases:

1. SafeSend is used to transmit updates, and no failures occur. This is the easy case.
2. Same, but some individual member fails while the rest of the group reconfigures and continues. Here, the challenge is that after a crash, your application will need to bring all the copies up to the same state. Since we are using a totally ordered multicast, this entails

checking to see how many updates each copy has applied, then fetching any subsequent ones, and then applying them in order. SafeSend helps in a limited sense: your crashed replica will have all the updates up to the point when it failed. But you'll have to handle the rest of the recovery protocol on your own, using a P2PQuery (a form of RPC) from the recovering process to some process that was in the group the whole time and has the full update list.

3. Last is the hardest case. Here the group needs to recover from a *total* crash in which the whole service fails. Our goal is similar to the one in step 2, but now we need to know that the delta includes any updates that any replica may have seen prior to the crash, *even if that replica isn't available at the time of recovery* (after all, Isis^2 groups have dynamic membership, and the recovered membership will presumably be very different from the membership prior to the crash).

The real role of the durability method is to help with this third case, because otherwise, you simply won't be able to solve it. After all, some missing updates might be available only from replicas that are still crashed. Where can we find copies?

This is precisely the role of the durability method. In effect, it maintains a totally ordered log of updates that are deliverable so that if we recover and have access to the log, that missing delta will definitely be available and can be applied (in our case, by redelivery) to the various replicas. Of course since Isis^2 has no way to know what each replica contains (which updates it reflects), this task falls to you as the developer: you'll need to filter out duplicated ones.

For example, imagine that update U reaches process P when the group membership was P,Q and that P's database gets updated. But now the power fails. Later we recover and are running with replicas Q and S. We can use the DiskLogger to recover and replay U so that Q and S will see it, and can do so immediately on recovery, so that the event ordering P saw will be mostly preserved. But we can't avoid the risk that we'll replay some updates that Q, S, or both have already seen. The best we can do is to make sure that they are presented in the right order, and that they have unique identifiers you can use to identify and filter them out. SafeSend thus can get us fairly far, but has limitations.

How exactly does the DiskLogger work? You'll need to start by initializing it, using a call like this:

```
myLogger = new Group.DiskLogger(g, “some file”);        // Must implement IDurability
g.SetDurabilityMethod(myLogger);
```

Every group member should do this call, if you plan to use the DiskLogger, since we can't predict in advance which members will play the Paxos “acceptor” role. Those never selected to perform the actual logging role will never create or access the file. Those that do play this role will append a suffix (an integer giving the rank within the subgroup of acceptor processes) to the pathname you provide, plus a “.dat” filename extension, and then will use this file for a log of pending messages.

Each time that SafeSend learns of a new message in a group member, the Isis^2 system will automatically call the CompletionTag ct = myLogger.LogMsg(Msg m) interface in the DiskLogger, during a first phase prior to delivery. The DiskLogger handles such a call by appending the message to the current log file and forcing it to disk. The message is now considered to be a pending candidate for delivery. The CompletionTag it returns contains a unique identifier for the message (the sender, viewid in which it was sent, and msgid which was initially assigned to it[4]). Once a sufficient number of logged copies exists (as determined by the SafeSendThreshold), the message can be delivered via upcall. At this point upcalls to your application code occur.

If your application is quick, it can simply do whatever update is encoded into the request and return, in which case we call myLogger.Done(ct) using the completion tag generated during the logging step. But some updates are slow to perform and for updates that take more than about a second, it is unwise to tie up the delivery thread. So, you'll need to spawn a separate thread. But now how can the DiskLogger know when the update is finished? Our solution is to let you access a unique id that can be used to tell the system when your update thread has completed its work. You'll need to do a call to CompletionTag ct = myLogger.GetCompletionTag(), and then a call to myLogger.BeginAsyncUpdate(ct) before spawning the thread. The ct object will encode a unique identifier for the message. Now you can simply pass the tag into the new thread. After it does its work, it should call myLogger.Done(ct), thereby informing the DiskLogger that the update is terminated.

The DiskLogger periodically checks to see if every delivery upcall has terminated with a matching call to myLogger.Done(). When this condition is reached, it can garbage collect the corresponding logged message. However, it might not do so immediately, and it will only garbage collect them in delivery order. Thus, if SafeSend delivers updates X, Y and Z in that order, the DiskLogger won't garbage collect Z until X and Y and Z are all completed, at all members where delivery occurred.

Now we can explain precisely what happens if a total failure occurs. As you know, when the group restarts from crash, an initialize runs. DiskLogger hooks itself to that mechanism, and will also get a chance to run. It uses a simple leader election scheme and the zero-ranked member of the restarting group will reload the pending message set from the log file and replay them into the group, as new multicasts delivered in the new view, but with the same "ct" values as was used originally. Your job is to check for duplicates, and do the non-duplicated updates, in order. Then call myLogger.Done(ct).

As a convenience to you, if no call is issued to myLogger.BeginAsyncUpdate(ct), and your code returns without calling myLogger.Done(ct), we'll do that call for you. Thus if your SafeSend request handler doesn't fork a new thread and simply runs without has been called (either automatically or by your code) in all members of the view within which this replay delivery occurs, the message will be garbage collected after all the updates are completed. Notice that if you were to call myLogger.BeginAsyncUpdate(ct) and never call myLogger.Done(ct), DiskLogger can never garbage collect anything and retains a full history of every message delivered in the group, in order. While this may sound like a useful feature, we recommend not trying this: it leaks storage and will cause restart from a total failure to run slower and slower, since every single update will get replayed in this case. If you want a log of messages delivered to your group, keep one on your own.

[4] Why use this triple and not just some sequential counter? The problem is that messages get logged *before* the sequential counter is actually assigned.

To detect and ignore duplicate updates, you can either encode some form of request id into your own application logic, or use our “ct” tags. The tag will have a unique per-message value and the same value is used even on replay.

A very interesting question to consider, and perhaps the last such question, is this. Recall our scenario from several pages ago, in which P and Q are supposed to perform U, but the group fails before both are done. Now suppose that we use OrderedSend to deliver U, but now call Flush before replying to external users. Under what conditions can this sequence we achieve consistency? What form of durability do we obtain?

The answer turns out to be particularly interesting: with this sequence we obtain exactly the form of durability needed in the first tier of modern cloud services, where applications are limited to maintaining soft state. In soft-state services, any reboot is always from a clean state; a restarted member obtains state via state transfer from some live member, or from a more durable source deeper in the cloud, such as a checkpoint file or a database living in the second tier. OrderedSend with a Flush thus gives us quite a strong property: consistency, and a form of amnesia-freedom (requests won’t be forgotten unless the entire service crashes); the guarantee is weaker than the durability property obtained with SafeSend, but on the other hand, SafeSend doesn’t scale well enough for use in the first tier. OrderedSend plus Flush, in contrast, definitely can be used in that setting!

The following table illustrates various options for actually invoking these methods. Variations exist to support different coding styles, particularly with respect to the collection of replies. Our own preference is to use the "delegation" style with inline code to handle the replies, but Isis^2 has the same behavior and performance in all cases.

Multicast operation

Send(request-id, args)

Sends a point-to-point message containing the "args" to the designated member of the group. The request-id is a pre-registered request identifier associated with request handlers in the group; args must have matching types. Variants exist for other ordering and durability properties: OrderedSend, SafeSend.

Send(dest-list, request-id, args) *or* **Send(request-id, dest-list, args)**

This variant form of Send can be used to send a multicast to a subset of the members of a group. There are two variant forms: the dest-list can appear before or after the request-id. In the latter case, the method invoked must have a first parameter that matches the dest-list argument type; in the former case, the method would not receive the dest-list as an argument.

The dest-list itself can be in either of two forms: a **List<Address>***, in which case the message will be sent to the specified addresses provided that they are live members of the group. The message will be sent only to the listed addresses.*

In the second form the behavior is a bit more complex and is intended only for use with the Isis^2 DHT and scalable aggregation mechanisms. If the destination is specified as **QueryKey<KT>(List<KT> keys)***, the system first confirms that the Isis^2 DHT mechanism is in use (an exception will be thrown if not). The list of DHT keys is mapped to create a list of the corresponding DHT shards. Then a single representative is selected per shard. The originator of the Send is always included in this list, even if the sender does not map to a shard referenced by the key list. The Query key can then be employed as the key in an aggregation operation that combines sub-results into a single aggregated result.*

Because the QueryKey itself will often be needed in the request handler, we've added a short-cut to avoid needing to specify it twice. If the request arguments start with a QueryKey, then Isis doesn't force you to also provide that same QueryKey as a "dest-list". Instead, the system checks the arguments and takes the QueryKey from there. Thus rather than calling Send(qk, request-id, qk, args…) you can just call Send(request-id, qk, args…)

Query operations

nreps = Query(nreplies, [timeout,] request-id, args, EOLMarker, r1List, r2List)

Multicasts the "args" to the group, waits for "nreplies" responses. The variable nreplies can be a constant (e.g. 1 or 3), or ALL, or MAJORITY. The optional timeout tells how long to wait and what to do if a response isn't received from some group member. The request-id is a pre-registered request identifier associated with request handlers in the group. Same for OrderedQuery, SafeQuery.

An object of type EOLMarker is used to separate args from reply vectors. This is a special Isis^2 type used purely as a marker.

If the optional timeout is omitted, we automatically use a 15-second timeout and a TO_NULLREPLY action.

The r1List, r2List etc are each objects of type List<T> for whatever type T you expect from the Reply (e.g. one might be a List<int>, another a List<Foo>, etc). Isis^2 will append replies to these lists in the same order, so that the I'th item in r1List is from the same sender that provided the I'th item in r2List.

For example, if rval1 will be of type int, r1Vec in the caller should be declared List<int> r1List = new List<int>(), etc. You can then use r1List.toArray() if you prefer to work with an Array rather than a List, although in fact the efficiency should be comparable in C#.

nreps = Query(nreplies, [timeout,] dest-list, request-id, args, EOLMarker, r1List, r2List)
nreps = Query(nreplies, [timeout,] request-id, dest-list, args, EOLMarker, r1List, r2List)

Same as Query but with a dest-list argument. (See discussion of Send)

Byte[][] reps = QueryToBA(nreplies, [timeout,] request-id, args)

Same as Query() but the replies are returned as a list of byte[] vectors, each of which marshalls data from some single group member. Data can be extracted as a vector of objects using Msg.BArrayToObjects(ba), or one can use the Msg.InvokeFromBArrays() API to request a callback into code that receives the demarshalled data as vectors of the corresponding object types. Variants exist for other ordering and durability properties: OrderedQueryToBA, SafeQueryToBA. The dest-list options work as they do for Query.

QueryInvoke(nreplies, [timeout,] request-id, args, (myDelegateType)myDelegate)

Same as Query() but the delegate you supply is invoked with argument vectors of appropriate types and sized to match the number of replies actually received. The delegate can be declared inline (as in the examples we've seen above) or can be a method declared elsewhere in your code, with the right type signature. Variants exist for other ordering and durability properties: OrderedQueryInvoke, SafeQueryInvoke. The dest-list options work as they do for Query.

A query won't be much use without the ability to send replies:

Reply operations
Reply(rval1, rval2, ….) *Allows a query handler to reply, sending result objects which must match the type signatures used in the process that issued the Query (type mismatches are detected and the system will complain about them). If sending user-defined objects, make sure that the client declares a class with the same marshaled data and registered using the same type code.*
RawReply(rval1, rval2, …) *Sends a reply unreliably. This API is for advanced use only.*
NullReply() *Allows a query handler to indicate that this particular member won't be sending a reply. Thus, even though the member is in the current view and received the Query, the reply vector will omit a response from it. Sent automatically if the request handler returns without calling Reply, NullReply, NoReply or AbortReply.*
NoReply() *NoReply is a tricky option and should be use with care. It specifies that the current member will not send any form of reply to the message (that is, not even a NullReply). While this saves message traffic, it cannot be combined with a Query that waits for ALL replies, because such Query would (always) time out. Isis itself uses NoReply in the implementation of SafeSend, which can be configured to wait for replies from Φ group members, but that code works in part because SafeSend automatically finalizes any multicast in the event that the group view changes. Thus, if you try to use this mechanism and the node that should have sent a reply fails, you could get into a situation where the view changes, but the Query initiator is still waiting for a reply. It will eventually time out and might kill healthy group members.* *We do offer a workaround for this case, but it requires a tricky style of coding. You can call g.SetReplyTo(Thread t) to specify an execution thread that will reply to the current request. This way, if a view change occurs and your code notices that the handler for some set of requests crashed, one or more backup processes can step in and reply on behalf of the failed node. But the needed logic won't be trivial to implement, and hence we recommend against using NoReply unless you feel that doing so is absolutely unavoidable.*
AbortReply("why this request must be aborted") *AbortReply allows a query handler to interrupt the caller, typically because the request was malformed in some way. Note however that because each member receives the query concurrently and separately, even if one does an AbortReply, others will still have received the same request and may be processing it. Their replies, if any, will be ignored. Immediately terminates the Query; other results (if any) are ignored.*

As noted earlier, there are some rules associated with who performs the Reply. When Isis^2 does an upcall to your code to deliver a Query, it also holds a lock that will prevent other message deliveries in the same group until the query handler returns. The handler is expected to issue a Reply, NullReply or AbortReply and if it doesn't do one of those things, Isis^2 sends a NullReply on its behalf.

You can fork off a handler for a request, by creating a new thread (Thread t = new Thread(delegate() { …. My code that will run in a new thread }); We recommend that if you do this, you give the thread a name, as in t.Name = "the thread doing the work for Query " + stuff; Then before you call t.Start() to enable execution, tell Isis^2 about this thread this way: myGroup.SetReplyThread(t); t.Start(). This disables the automated sending of a NullReply and tells Isis^2 that eventually, thread t will call Reply, NullReply or AbortReply. (Don't forget to do so! Isis^{2} won't check and failing to do these calls would cause all sorts of problems over time).

Monitoring Group Views and Members

To monitor a group so that new views will be reported, just attach a view monitor:

```
myGroup.ViewHandlers += (Isis.ViewHandler)delegate(View v)
{
   Console.WriteLine("myGroup got a new view event: " + v);
};
```

The fields of the View structure list a unique identifier for the new view, the members in this view (in the order they joined the group), and then as a convenience, list the most current view "delta", consisting of members who joined (the "joiners") and members that departed ("leaving"). No distinction is made between members that crashed and members that voluntarily left the group. The code above will pretty-print the view structure on the console.

You can also watch a member of a group. If a future View becomes defined in which that member's status changes, a callback will occur. As seen below, *who* is the address of the watched member, and the event indicates what happened: the value will be W_JOIN or W_LEAVE.

```
myGroup.Watch[who] += (Isis.Watcher)delegate(int ev)
{
   Console.WriteLine("myGroup was watching " + who +", when event " + ev + "occurred");
};
```

A watch can also be cancelled, as in the example below:

```
      myGroup.Watch[who] += myWatcher;   // Set a watch on process 'who'
                 . . .
      myGroup.Watch[who] -= myWatcher;   // Cancel it

public void myWatcher(Address who, int ev)
{
         Console.WriteLine("myGroup was watching " + who + ", when event " + ev + "occurred");
 };
```

Peer to Peer Communication Between Group Members

Some applications require ways for group members to issue calls directly to other members. For this, Isis^2 offers a P2PQuery that queries a specific member of a group, returning a single reply. The basic syntax is identical to the options for the standard Query APIs. Ordering is FIFO on a per-sender basis. a packet is lost on the network, Isis^2 automatically resends it until it is acknowledged by the receiver's Isis^2 library. There are no limits on the sizes of the messages that can be sent.

P2PSend(member, request-id, args)

Sends a message to a group member, point-to-point, but in a sequenced and reliable way.

byte[] reply = P2PQueryToBArray(member, [timeout,] request-id, args);
int nreplies = P2PQuery(member, [timeout,] request-id, args, EOLMarker, r1List, r2List);

Same, but waits for a response from the target member. The optional timeout tells how long to wait and what to do if a response isn't received. The request-id is a pre-registered request identifier associated with request handlers in the group. In the second form of P2PQuery, the EOLMarker is used to separate args from reply vectors. Note that even though only a single reply is expected, r1Vec, r2Vec, etc are each of type List<T> and should be provided as initially empty lists, into which the replies can be saved; if there is no reply, the lists will be empty after the call; otherwise, the I'th reply is is converted to a list of objects of the types given in the Reply() call, and each object is appended to the corresponding list. Thus, the I'th element in each of these lists came from the same sender: the one that generated the I'th reply that was received by this caller.

Throws an AbortReply exception if the request-id you specified corresponding to a group request handler that doesn't allow upcalls from clients. nreplies will always be 0 or 1, since the reply comes from a single group representative.

If the representative leaves the group while this request is underway, or fails, the request is reissued. This can cause a single request to be delivered more than once, if a race arises in which the representative leaves or fails "while" processing a request. The application should be designed to tolerate that sort of reissued requests.

RawP2PSend(member, request-id, args)

Sends an unreliable point-to-point message to a designated member of a group. No acknowledgments are send and the message will not be resent if lost. Useful in building gossip protocols.

int nreplies = RawP2PQuery(member, [timeout,] request-id, args, EOLMarker, r1List, r2List);

Sends a RawP2PSend and then waits for a reply. The Reply can be sent using Reply() or via the unreliable RawReply() API.

g.AllowClientRequests(request-id).

Performed within the group members to enable upcalls from clients to the handlers associated with this request-id. If a request arrives from a client and the request-id doesn't allow client requests, the message is silently ignored if it was a P2PSend (which is why we don't recommend using that API), and triggers an AbortReply if it was sent as a P2PQuery.

g.RedirectClient(Address client, Address newRep).

Issued by a representative to shift this client to some other group member which will become its new representative. Since some requests may be in the pipeline at the time this call is done, a few more requests might still be received by the old representative before the Client has switched to the new representative.

State Transfer

When you join a group, the joining member will often need to catch up with the existing members. Isis^2 can help if the state is small, but if the state is large you need to do this in two stages. The same methods also play a key role in making groups *persistent* (see below).

For large state, we recommend that you query the group and read as much state as possible in a background loop before attempting to join. This may be slow (in fact, you should design it to be slow, so that you won't disrupt the normal group operations), but it should allow the joining member to be nearly caught-up before it actually issues the join request. You'll need to somehow track the "time" associated with the state to make it work, for example the number of updates that your copy of the state reflects.

If the group state is small, or if the "delta" that remains to be transferred is small (maybe, just the updates the group has seen since update-time 21991), you can use "state transfer". Here's how it works. First, it may help to look back at those first figures and remind yourself of what state transfer is trying to accomplish: these were the white arrows inside the blue ovals in our group execution pictures. The basic idea is simple: when a new member (or members) is to be added to the view, Isis^2 first does a Flush operation on its own, just like the ones you can invoke directly. Then it invokes your code, which you provide this way:

```
delegate void loadichkpt(int stuff);
delegate void loaddchkpt(double stuff);


    . . .

    int myInt = 12345;
    double myDbl = 987.654;

    g.MakeChkpt += (Isis.ChkptMaker)delegate(View nv)
    {
      // Send a single integer
      g.SendChkpt(myInt);
      // Send a single double
      g.SendChkpt(myDbl);
      // Done
      g.EndOfChkpt();
    };
    g.LoadChkpt += (loadichkpt)delegate(int i)
    {
      Console.WriteLine("Got integer checkpoint = {0}",  i);
      myInt = i;
    };
    g.LoadChkpt += (loaddchkpt)delegate(double d)
    {
      Console.WriteLine("Got double checkpoint = {0}", d);
      myDbl = d;
    };
```

As seen in this example, you implement a state transfer function by taking a few simple steps before issuing the Group Join() request. First, define a pair of methods, one to send the Group state, and the other to receive the state. Then register those two methods. The sender transmits the state as a series of “checkpoint” messages, each of which is delivered in order and triggers an upcall in the joining process. Finally, call the end of checkpoint method.

Above, we illustrated the basic idea for a group that has a state consisting of an integer and a double, which for purposes of illustration we sent in two messages. Note: if the objects are small it would be better to send these two pieces of data in a single message, because messages cost time and larger messages are much cheaper, on average, than multiple small ones!

Notice that the delegates don’t actually run when they are declared. The make-checkpoint delegate runs in the group leader (the oldest group member) when Isis^{2}is defining a new view for a joining process, and in fact the argument “nv” let’s you see who that process happens to be (this is important if a process pretransfers some of the state: it lets you keep track of what you already sent it). The loader methods run in the new process just as the join is occurring, and immediately *after* the last checkpoint message is received, the New View upcall will occur in all group members, if you happen to be using that feature.

Our code might worry the reader: the checkpoint upcalls will occur in a different thread than the thread doing the Join(). This happens to be safe because the Join() will be paused at the time the upcalls occur. However, one should certainly keep in mind that there are potential concurrency control conflicts *any time* Isis^2 issues an upcall to your code, and while these two examples happen to be correct, in general one would need locking.

Checkpoints are always sent by the rank-0 member of the view in which the new member(s) are added, and a single checkpoint is sent even if there are many joiners (they each get a copy).

Initializing and Terminating a Group

A group can have an intializer, which will be called only if there are no active group members already running (in that case a state transfer would be done), and only if there is no persistent checkpoint associated with the group from some previous period of activity (in that case, the checkpoint would be loaded as if a state transfer was occurring, although it will be done from the persistent storage file and not from some member). In the case where initialization *does* need to occur, Isis^2 calls:

```
    myGroup.Initializer += (Isis.Initializer)delegate
    {
      Console.WriteLine("myGroup is restarting from scratch!");
    };
```

Persistent Groups

You can associate an Isis^2 group with a stored checkpoint. This allows the group to maintain its state across periods when all members exit.

The API is simple. After creating the Group object and registering types and methods and aggregators, immediately before issuing the Join() or Create() request, you simply need to invoke myGroup.Persistent("some file name"). The file name must be accessible to the calling process, which must have permissions to create, read, write and replace the file. This file should be located on a globally accessible file system, so that a single persistent file is used by all members (only the leader of the group, namely the rank-0 member, will actually update it). The caller must also have permissions to create a temporary file in the same folder (to avoid corruption if a node crashes while making a checkpoint, Isis^2 first writes the entire checkpoint to a temporary file, then uses the Replace() system call to atomically replace the old file with the new version, leaving the most recent previous version in the same folder with the name "name.bak" just in case you ever wish to revert).

Notice that even though your group will have multiple members, it still will have just one checkpoint file, and only the leader will actually do updates at any point in time. However, the role of being the leader depends upon the current membership view. This is why all members need to do a call to Persistent, and why all must be able to access the checkpoint file (which, obviously, will have to be on a global file system).

Isis^2 will create the persistent storage file the first time it creates the group (at the same point as it calls your group *initialization* method). It will store checkpoints into the file from time to time, as explained below. Later, when restarting the group after a complete shutdown, if the persistent file exists and is non-empty, the checkpoint in the file will be loaded. In this case your initialization method will *not* be called; the process recreating the group will, in effect, receive a state transfer from the checkpoint, which was in fact created as a state transfer from the group leader in some previous incarnation of the group.

The checkpoint is created using the exact same method as we use to perform a state transfer. The state transfer method is called in the leader, and it generates a series of state transfer messages by calling myGroup.SendChkPt() one or more times, and then signals the end of transfer by calling myGroup.EndOfChkPt(). These state transfer messages are byte-serialized and written, record by record, into a temporary file. When the file has been fully written, the .NET "Replace" method is used to atomically replace the old persistent storage file with the new temporary one. As noted, this same operation renames the old version as a backup, deleting any older backup version.

Checkpoints are created at times convenient to the group leader. The leader does this either by calling myGroup.MakeCheckpoint() at a good point in the execution, or by calling myGroup.SetCheckpointInterval(int delay). (Note: You can switch back to manual checkpointing by calling myGroup.SetCheckpointInterval(-1)). MakeCheckpoint creates a checkpoint immediately; SetCheckpointInterval makes a checkpoint every "delay" seconds. Use the MakeCheckpoint method if your group has better and worse times for creating checkpoints (for example, perhaps you have an internal data structure cleaning and compacting method and only want to checkpoint the state when it has just finished running, or perhaps your application runs a series of calculation phases and you want to checkpoint only between phases, not mid-way through a phase). MakeCheckpoint() is also the only option if you want to revise the checkpoint after every update (an expensive option). Use the timer driven approach if you simply want to limit the data loss exposure so that a crash of the whole data center will not lose more than the last delta milliseconds worth of updates.

Note: When creating a checkpoint, keep in mind that an Isis^2 group address may be treated as persistent data, but that it is not safe to consider a process address as persistent.

Secure Groups

The group security features of Isis^2 are integrated tightly with the group persistency features. To use the secure groups interface, you should call myGroup.SetSecure() prior to calling myGroup.Persistent(). If you call myGroup.SetSecure with no argument; a random 24-byte AES key will be assigned; an overload of myGroup.SetSecure(byte[] AESKEY) allows you to specify the group key. In this mode the system will encrypt the data portion of each message prior to transmitting it, or before logging a checkpoint. If SetSecure is not specified for a particular group, messages between members of that unsecured group will be sent "as usual", in unencrypted form.

There is an important difference that the user should be aware of between the two versions of myGroup.SetSecure().

If the developer *does not* specify the key to use, then Isis^2 has a small problem: how will processes joining a group learn the group key? In this case, the group security scheme makes use of the platform file security. The *group key is stored in the persistent checkpoint file, and the data in the file itself is not encrypted.* In effect, we assume that only users who have permissions would be able to read these files. For example, suppose that your group maintains "myCorporateSecrets". You might create a permissions group on Linux or Windows for individuals permitted to access the secrets, creating the Isis^2 persistent store for the group (with the secret part of the SSL key in it) and then setting the access control list for the file to only permit access from processes running under the corporate secrets group. Processing would need to run under this account to be able to Join or Create the group. However, anyone with root (administrative) access to the file system can read any file and hence access the checkpoint contents, and the group key in this case.

In contrast, if the user specifies the key to use, the key itself *is not stored into the checkpoint file* (zeros are stored at that location instead), and *the checkpoint contents are encrypted prior to saving them, and decrypted on loading them.* This means that the user has responsibility for storing the AES key outside of Isis^2 and somehow obtaining it before a process calls myGroup.SetSecure(key). For example, a certificate store could be used. This means that *even a person with root access to the file system would be unable to access file contents or decipher messages on a wire.*

On the other hand, a user with administrative rights and a copy of the Isis^2 source files could still break this stronger scheme, by using a debugger to attach to a process that has joined a secure group and freezing the process mid-execution. In such cases the group encryption key is stored in memory and could be copied out using the debugger memory-inspection features, at which point the attacker could use the key to decipher checkpoint file contents or other communication.

You can also secure Isis^2 itself, which will prevent applications from attaching themselves to the that version of Isis^2 unless they have the security key. In this mode all messages sent or received are signed with a SHA2-256 hash that we encipher (just the hash, not the message) with the ISIS_KEY you specify through the environment variables used at runtime. Thus if you combine this ISIS_KEY flavor of security with the per-group version, the traffic on the wire is doubly-protected: all "system" messages are protected by cryptographic signatures, while all traffic "within" the group will be enciphered before transmission.

A third option is planned for the future: encryption using hardware-assisted methods in which the key resides *outside* the system, in a special trusted hardware module (a so-called TPM). Once this option is supported, if it is used, even an attacker with root permissions and source code for Isis^2 would be prevented from seeing checkpoint file contents or stealing the key. Yet even so, such a user could employ the debugger to examine data in memory with the application. Unless that data were kept in enciphered form *at all times* this would represent a security risk, for developers concerned about that sort of ultimate attack model. The difficulty, obviously, is that computing on enciphered data is non-trivial and only some operations are even feasible. Moreover, some computing models can leak data even if the data isn't deciphered for the computation.

Thus, the developer of a secured group must be sophisticated, must think about the attack model, and must be reasonable about both goals and methods used to achieve them. Isis^2 offers options but they don't address all possible goals.

How secure are Isis^2 groups, with these features in use? The bottom line is that the system is extremely secure, and yet that there are obviously limits. For example, a distributed denial of service attack on a group could easily trigger the system's failure detector, hence an attacker could probably cause a form of costly churn. While it wouldn't be possible to see data for a secure group on the wire, or to join such a group without knowing its secret key, within a modern data center many applications execute in virtual machines. These are sometimes copied around the network or stored on disk. Thus an attacker who manages to get his hands on a VM image of a running system could examine the binaries to extract group keys, or examine in-memory data. Moreover, the whole security architecture depends on the security of the keys we use: leave a key lying around and all bets are off. All in all, this is probably as strong a security model as one can find in a deployed distributed platform today, but as our remarks illustrate, if one wanted to sit down and disrupt a secure Isis^2 group, and had access to the data center network, it wouldn't be impossible to do so.

A final remark: Isis^2 protects itself against random noise using SHA2-256 hash, which it appends to each message. An incoming message lacking a valid hash will be ignored silently. But an attacker might know how to fabricate a message with a valid hash but an invalid internal structure, causing Isis^2 to crash. You can improve protection against this by setting the system parameter ISIS_AES to a secret key; Isis^2 will then encrypt the SHA2-256 hash (hence only an attacker with a copy of ISIS_AES would be able to construct a message that might slip through). Obviously, however, use of this feature depends upon maintaining the security of the ISIS_AES key.

Non-Member *Clients* of a Group

An application can create a form of stub to connect with a group by creating a new Client("name") object. The Client API permits the application to interact with a "representative" that can perform actions on its behalf.

A Client of a group must be compatible with the group: it must use the correct arguments for group multicast or query methods it plans to invoke, and must declare the identical types that the group used including using the identical type ID numbers (e.g. if the group members expect you to send in a PurchaseOrder, and the group uses TID=6 for this type, the client must also declare a class with the same marshaled data and registered using TID=6, just as the group members did).

Security considerations dictate the limited functionality of this Client interface. A client doesn't have the group key, and for that reason, isn't subjected to the normal Isis group authorization functions. Instead, the assumption is that the representative will filter requests, performing them in a safe and secure manner and doing so only if the client is one that should be permitted to do so. Thus rather than allow the client to Send to the group, the representative will receive requests in a point-to-point way and can make decisions. If a multicast is needed, the representative would need to issue the Send on behalf of its clients. Indeed, the client API doesn't even offer a Send capability: the only options offered are forms of point-to-point query.

These queries won't even reach the group without *explicit permission* from the members: By default, Isis^2 *blocks* calls from non-members of a group to the group handlers. To enable client calls to a particular handler, the group members must call g.AllowClientRequests(request-code). If this call is not done, and a message shows up from a client in the group, an AbortReply() will be automatically be generated ("The group has not enabled client requests to request-code *k*"). This will trigger an exception in the client.

The Client API offers the following three operations: P2PQuery, P2PQueryToBA, and P2PQueryInvoke. All are remote procedure calls that send a single request to a single process in the group, which takes some action, and then sends back a reply. Notice that the API lacks a P2PSend operation: to multicast, you should send a "Query" that requests for some member to resend your message, and then to acknowledge that the operation was performed.

A few words about load balancing: the client API selects the representative in an automated manner that spreads the role of representing clients around, but doesn't "load balance" per-se. In fact, Isis^2 has no idea what the loads within your groups are. Any given client will be assigned to some group member in a round-robin way, and once assigned, the representative will remain fixed until either that member leaves the group or fails (in which case a new representative will be assigned automatically), or the group member calls g.RedirectClient(Address client, Address newRepresentative). This API allows a group member to redirect this client to be represented by some other group member. The g.RedirectClient call can be done at any time but may not take instant effect, since requests could be in the pipeline when the request is issued.

In summary, then, the Client API allows a non-member to issue requests to a representative within a group, and if that representative has authorized client requests to the handler the Client invoked, its handler methods will be invoked with the arguments you provide in the or P2PQuery operation, and the response it returns delivered back to you either for storage into in-line data objects (they need to be declared as vectors even though only one reply will be received, to deal with the possibility that no reply is received due to failure, and also to work around a limitation imposed by C# involving passing object references into a procedure with a variable-length argument list), for delivery to you as a byte[] array, or for invocation of inline logic.

Often, the representative will perform a multicast Send or Query within the group on behalf of the client. The representative could, for example, relay a request, collect the response as a List<byte[]> object, and either pre-process the results into a single collective answer, or just sent the vector of byte vectors back without further action, for the client itself to process (you would need to sent the answers as a byte[][] array because the Isis^2 message layer doesn't currently support sending List objects).

When relaying actions that can change the state of the group, caution must be taken. A limitation of the client API is that in the event of a failure, the client won't know if the operation was performed: the P2PQuery will return 0 replies, but the representative might have had time to relay the request before it crashed, hence the group may already have performed the request. If the client re-issues its request, it would be performed twice. For this reason, we recommend that the Isis^2 API be used for requests that are "idempotent", in the sense that repeated execution of the same operation won't cause problems, and will return a sensible answer. For example, one can query many databases again without harm, if a first try fails. Relaying updates can be safe if the updates can be reapplied without causing harm.

If an update ***can't*** be reapplied and exactly-once behavior is required, the Client API may still be useful, but you'll need to implement additional logic to turn the non-idempotent request into one that behaves idempotently. For example, if the client of a group sends a request that will give a raise to employee 1716, it is going to be important to not give that raise twice just because the client's representative happens to crash just as the action is being performed. As the developer, you would need to add a unique request identifier to the update, log these when the action is performed, and not perform the update if the action duplicates something already in the log. The issue is a familiar one seen in any transactional database system, and we recommend that the developer consult any good database textbook to learn more about options for solving it. Isis^2 could also be used in conjunction with a transactional package that addresses those issues: You would use Isis^2 for distributed computing, and the transactional solution to manage the underlying data.

Registering New Data Types (Classes)

Isis^2 needs to be able to marshall the objects transmitted in Send and Query requests into and out of its underlying messages, and it does this by converting them to a very compact byte coding form. Each type is represented internally by a byte-code; values 0..127 are reserved for the user, and values 128...255 for Isis^2.

The usual base types are built in and supported directly: int (and all the variations: int16, int32, int 64, unsigned), float, double, byte, char, string, etc. We also support one-dimensional arrays of all of these types and two-dimensional arrays of int32, float and double. Isis^2 can also marshall its own data types: Address, View, etc.

Users are welcome to declare data types of their own. To do this, one must "register" the classes that will be used, and each registered class needs a method that will encode the data in the fields of an object of that type into a byte[] vector, and then decode a byte[] vector to construct a new instance of the desired type. Here's a simple and automated way to do this:

```
[AutoMarshalled]
 public class GRPair
 {
    // These fields will be included in the outform representation
    public Address gaddr;
    public int rate;

    // This field will not be included in the outform representation
    internal bool refresh = false;

    // Null constructor: Needed for AutoMarshaller
    public GRPair()
    {
    }

    // Used by the application to initialize a new GRPair object
    internal GRPair(Address ga, int r)
    {
       gaddr = ga;
       rate = r;
    }
 }
```

In our example, the designer wanted Isis^2 to transmit every public field in the class GrPair. So he or she attributed it with the [AutoMarshalled] attribute, as seen right in front of the class definition line. Next the designer needs to call Msg.RegisterType to register the type:

```
internal const byte TID = 123;
   . . .
Msg.RegisterType(typeof(GRPair), TID);
```

TID is a unique type identifier that must in the range of 0 to (current limit) 127. Notice that even though the AutoMarshalled attribute was specified, we still require a call to Msg.RegisterType and it must occur before you join any groups.

Isis^2 will scan an Automarshalled class definition searching for public fields (non-public fields are ignored), encode those field using its marshalling functions, and place the resulting data into the message. When demarshalling, Isis^2 creates a new empty instance of the object type (hence the need for a public null constructor) and then fills in the public fields, in order.

Sometimes, of course, a class has a mixture of public fields you don't wish to transmit or initialization logic that can't be executed in the null constructor because the public fields haven't yet been assigned values when the constructor runs. In these cases you need to define methods that do the required data marshalling and demarshalling by hand, using the built-in Msg.toBArray() and Msg.BArrayToObjects() methods:

```
public byte[] toBArray()
{
   return Msg.toBArray(gaddr, rate);
}

public GRPair(byte[] ba)
{
   object[] obs = Msg.BArrayToObjects(ba);
   int idx = 0;
   gaddr = (Address)obs[idx++];
   rate = (int)obs[idx];
}
```

Note that these methods must be public, which implies that the class itself must also be public. Otherwise, C# scoping rules prevent Isis^2 from seeing the methods, or the fields, that need to be converted. Further, notice that when this form of explicit marshalling is used, you can easily employ the more standard C# object serialization methods; these produce rather slow code and verbose byte form representations, but they work in a very general way and can be used to marshall extremely complex objects. To use built-in C# serialization, employ the C# BinaryFormatter, MemoryStream and the Serialize and Deserialize methods to create a byte vector that can be sent and later used to reconstruct your object.

In Isis^2 v1.x.xxxx, since application-declared data types are defined on a per-process basis, an effort is made to confirm that the members of a group have the same signatures and this occurs when a process is joining the group. *All applications that join a group must declare all the event handlers for the group, with the same types that every other member declared, and using the same type ID values as they used.* In effect, different processes can join different sets of groups, but members of any given group must agree on the type signature for that group, which includes the set of methods handling group events and the types associated with them, and the byte-code identifiers for those types. This way, when member A sends an UPDATE, member B is able to correctly interpret the message and deliver it to the a handler that agrees on the type signature for that action.

Warning: Our type checker isn't as sophisticated as it really should be and you can trick it; the resulting code will be type-safe, but will throw a runtime type exception while the application is active (where it can be confusing and hard to debug). The issue is that our current type signatures for group methods are "shallow": we confirm that every member defines the same event handlers and uses the same types for the parameters, but we don't transmit signatures for those parameter types. Thus if the same parameter type is defined differently in different members but using the same type names, Isis^2 won't catch this at group join. Instead, it will get a surprise later, when an object of that named type turns up but doesn't match what was expected.

For example, suppose that everyone in a group defines a method for request-id UPDATE, and everyone agrees that the method receives an object of type Foo. But in member A, Foo is defined as an object containing an int and a double, while member B defines one of those fields as an object of type Bar. Ideally, Isis^2 should prevent one of these processes (whichever shows up last) from joining the group since Foo means different things in them. Instead, however, the system will mistakenly assume that all Foo's are identical because they have the same name and will allow the join.

The problem arises later, when an update arrive. The system will try to invoke the UPDATE handler in each member, using a reflection method in which the types of the arguments match the types of the objects in the message. A member that receives a message containing data for a Foo but has a different Foo definition will throw a runtime type exception when trying to instantiate the local Foo object instance by demarshalling the message. Worse, this won't be easy to understand when you examine the stack trace, since the exception occurs in the IsisLib callback layer, not in your code: we're "about" to invoke your Foo constructor (or in the midst of the AutoMarshall code), but the fault actually occurs while still in Isis^2.

In some future version of Isis we plan to fix this by walking the inner type definitions recursively, and including those in the transmitted type signature. Until then, users should be careful that any given type has a single application-wide meaning, and a single application -wide type ID.

MultiGroup APIs, Barrier Synchronization

Applications working with large numbers of groups may find it convenient to use the Isis^2 multiGroup API, which offers methods for joining or leaving multiple groups in a single atomic action, and even permits a process to add or remove sets of processes from sets of groups, all in a single call to Isis^2: Group.MultiJoin(), Group,MultiCreate(), Group.MultiLeave(), Group.MultiTerminate(). To synchronize with the sets of processes involves, the master calls BarrierWait() and the workers processes call BarrierReached(). These APIs aren't hard to use, but they pose a few small planning issues for you as the designer. We discuss them further below, in the section that describes BatchStart. Note that BarrierWait/Reached can be used in any setting you wish and are not limited to this startup situation.

Controlling Multicast Data Rates

As mentioned above, sometimes you'll need to provide our built-in flow control mechanisms with a bit of help to avoid patterns of bursty transmission at high data rates that might overwhelm a group. We do our best, but if your application manages to fool Isis^2, there may be no alternative except to limit the rates at which your group members send multicasts.

The method myGroup.SetRateLimit() can be used to specify a multicast sending rate limit that Isis^2 will enforce on your behalf. The limit is in units of multicasts per second and Isis^2 computes the current rate by averaging over the past two seconds on a per-sender basis. Thus, if a process X calls myGroup.SetRateLimit(20), the limit in question applies to X but not to other group members, and X will be limited to an average rate of about 20 multicasts per second. An overload of myGroup.SetRateLimit() allows the user to specify a second limit in terms of bytes per second that can be transmitted, as well as a limit on the message rate.

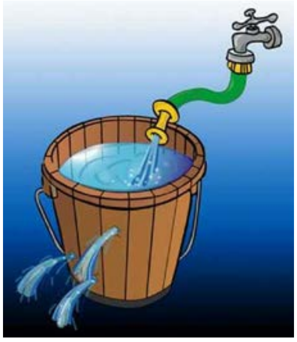


Figure 2: Leaking bucket rate controller.

The actual rate can be briefly higher, or briefly lower, and is implemented using a "leaky bucket" technique, in which tokens are generated at the designated rate (20 per second), and each outgoing message requires a token to be transmitted. Unused tokens age out of (leak from) the bucket after 1 second.

For example, suppose that you specify a message rate of 20/sec, as above and leave the byte rate at infinity. Each second, Isis^2 will add 20 "tokens" to the output buffer for the group. Assume that the group is currently idle. After 1 second, Isis halves the residual count (so the 20 "old" tokens become 10) and adds 20 new ones. Thus, in an idle state, there will be 20 + 10 + 5 + 2 + 1 or 38 tokens available. The halving process is intended to mimic the notion of a bucket with a leak.

Now suppose that your system generates a powerful burst of multicasts over a sustained period. The first 38 outgoing messages find tokens in the group and are sent with no delay: each message needs one message token, and for each byte in its marshaled form, it will consume a "data" token. The 39th multicast is forced to wait; it pauses until a new batch of 20 tokens are added to the group. Now the 39th multicast can be sent (as can the next 19 messages). The steady state will be limited to 20 messages at a time, although notice that these are potentially transmitted as bursts of 20, each time tokens are added to the group.

If needed, you can smooth out these bursts. If our 20 tokens were added 4 at a time at 200ms intervals, we could get a much smoother effect, in part because we're feeding tokens into the rate limiter at a steadier pace, and in part because the number of old tokens lingering from past idle periods would be smaller (at most 3= 4/2 + 4/4). Note: The smallest internal permitted is 50ms.

The myGroup.SetRateLimit method has 3 parameters: the message rate in messages per time unit, a data rate in bytes per unit time, and a time unit. The default message and data rates are infinite and the default time unit is 1000ms. If SetRateLimit is never called for a group, the leaky rate limiting mechanism will be disabled and the group won't incur any overheads at all.

Limiting Which Groups can Use IP Multicast in Isis

The Dr. Multicast algorithm (discussed in more detail later in this document) is controlled by two parameters, which can be set through the “environment” variables, which are scanned by Isis^2 at startup. These variables are:

```
bool ISIS_UNICAST_ONLY = false; // If true, Isis uses no IPMC, but still uses UDP

string ISIS_HOSTS;              // Location of Isis Oracle in UNICAST-only mode
string ISIS_NETWORK_INTERFACES; // Network interfaces Isis is permitted to use

int ISIS_MCRANGE_LOW =   5000;  // Isis allocates IPMC addresses in this range.
int ISIS_MCRANGE_HIGH =505000;
int ISIS_MAXIPMCADDRS = 25;     // Limit on how many IPMC addresses can be in use.
```

Although Isis^2 can run with very few IP multicast addresses, or even none at all, doing so isn’t trivial and isn’t necessarily a good idea. To shut off all use of IP multicast, set ISIS_UNICAST_ONLY to true, but then tell the system how to find the Oracle (see next subsection) using ISIS_HOSTS. Notice that in the default configuration, the system uses a 500,000 group “address space” for IPMC addresses. This is selected to be large because Isis^2 can malfunction if two groups accidently map to the identical group address, a step done using a hashing function (the groups would, in effect, be merged into one, like it or not).

For ISIS_UNICAST_ONLY, the system will just map all multicasts to UDP sends. This is done in a way that tries to be efficient (we map groups to a kind of tree structure, so the sender sends out perhaps 10 UDP sends even for a massive group, with the receivers relaying the multicasts and so forth) but of course will be slower than just using IPMC if we’re permitted to do so.

You can also pin Isis2 to specific network interfaces. There are a number of options for doing this. We favor setting ISIS_NETWORK_INTERFACES to the interface names Isis is permitted to use, like “ib0”, or “et0,eth1”, etc. But we also have a feature ISIS_SKIP_FIRST_NETWORK_INTERFACE for use on DETERLAB and EMULAB, where the system is supposed to ignore the first network interface on a given cluster (no matter what name it has), and bind to all others. By mixing and matching you should be able to configure the system in any way you wish. Of course, it will break rather dramatically if you use these configuration options haphazardly or combine them in ways that don’t make any sense at all.

Warning: Dr. Multicast periodically recomputes the mapping of virtual to physical IPMC addresses and during this period, while groups are switching to their new addresses but old ones have yet to time out from the data center network, the number of IPMC addresses in use may briefly be as much as double the limit expressed in ISIS_MAXIPMCADDRS.

One further comment. On machines that run software derived from Emulab to manage large clusters, there will typically be a set-aside management network on which IP multicast is possible but discouraged. Emulab requires that this network correspond to the first network interface on each machine (in an Emulab cluster, every machine has at least two network interfaces). Set the environment variable ISIS_SKIP_FIRSTINTERFACE=true to warn Isis not to use the first network. Set the environment variable ISIS_NETWORK_INTERFACES to a list of interfaces if you wish to take full control over the choice of network interfaces on which Isis^2 will use IP multicast. This should be a list of the names of the interfaces Isis should use, taken from the names printed by ifconfig –a (or ifconfig /all on Windows). For example, you might set ISIS_NETWORK_INTERFACES=ib0 or ISIS_NETWORK_INTEFACES=eth0,eth1.

Can Isis^2 do MapReduce? Can it do Things MapReduce Can't?

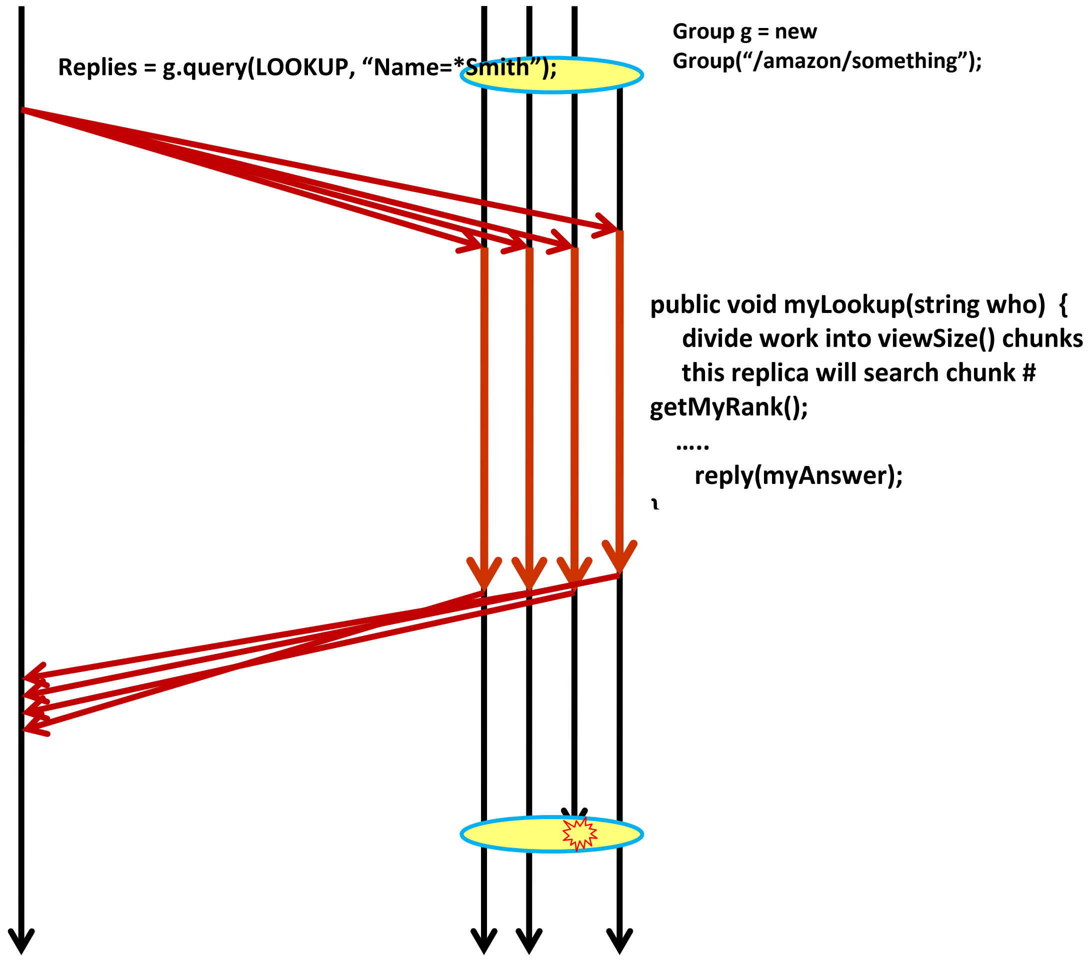
If you know much about cloud computing, you know a lot about MapReduce, Hadoop, Dryad and the many other tools for doing really massively parallel computation in cloud settings and clusters. MapReduce starts by spreading a task over a set of nodes: the "map" step. Each gets part of the big job, whatever that might be. Next, we do some form of "reduction", combining intermediary results to squeeze the data down. We could Map to a bunch of nodes and Reduce just to a single node, but sometimes we Map to many and Reduce to many too, and then run new MapReduce actions on the reduced data. And from this basic pattern, we end up with Web Indexes, information about which products people buy after looking at Sony Wide Screen TVs, which web pages are the most authoritative source for information on herbal teas to treat baldness (sadly, they don't work), you name it.

Isis^2 can support a style of execution very similar to what one achieves with MapReduce, although there are some differences: with MapReduce the user specifies a problem as a set of tasks; with Isis^2 the problem is usually posed as a query and the group members break it into subtasks, with each taking responsibility for part of the query (normally, using their rank to decide who does what). The mechanism most closely related to MapReduce is the Isis^2 DHT (key-value store), especially when combined with the .NET LINQ features and with Isis^2 aggregation. But in fact MapReduce-style computation is common within all of Isis^2 and can be initiated even without using the DHT.

Where Isis^2 goes beyond MapReduce is in its ability to provide consistency for responses to operations that involve the whole group, even when updates may be occurring at the same time as the query, and even when failures occur. MapReduce only provides guarantees for underlying data that is static, and indeed the entire MapReduce model is basically functional: tasks that can be computed again and again, if desired, with identical results. Isis^2 is far more suitable if you are dealing with evolving data and yet want consistent results.

For example, when doing a query with the Isis^2 using the strongly consistent DHTOrderedPut and OrderedQuery or OrderedSend followed by Aggregation approaches, you gain the certainty that each data item in the group – each key-value tuple – was considered exactly once and that the result includes all contributed values. Moreover, if data was evolving when you did this, you have the certainty that the query occurred at an instant when the group wasn't applying an update (along a "consistent cut". And while we can't avoid the risk of a disruptive failure, Isis^2 will signal that a query was disrupted, so you'll know, and also can automate the healing of the group itself.

The basic model covering all of these cases is this: The application sends a query of some sort to the members of a group, and they each do part of the work and than each replies to the caller, who collects those requests and reduces them into an answer of some sort. End of story and quite parallel too, but obviously only for a group with at most a moderate number of members, like 10 or perhaps 100. With tens of thousands of members, that 1-to-all and then all-to-one model stumbles because the all-to-one step can overwhelm the query sender. Like this (we've turned the picture sideways now, with time running top to bottom, just so that we could add text to the figure):



```
Group g = new
Group(“/amazon/something”);
```

```
Replies = g.query(LOOKUP, “Name=*Smith”);
```

```
public void myLookup(string who)  {
    divide work into viewSize() chunks
    this replica will search chunk #
getMyRank();
   .....
     reply(myAnswer);
}
```

Figure 3: A client of a group queries its members, which compute the reply in parallel.

Above, the query was issued by the process on the left. In this example that process is actually what we would call a *client* of the group: it isn’t a member, but can talk to it anyhow (and we should mention up front that the picture lies: when you talk from a client to a group, your request gets vetted and then relayed by some *representative* member, who could also decide that you are unworthy of talking to the group and reject the request (perhaps, you didn’t know the secret handshake). The representative would also relay the answer back to you. But this is all detail and doesn’t impact the overall idea very much.

Anyhow, the request – a LOOKUP in this example – reaches the group members, and they each perform some part of the task. Our example is oversimplified; the details of the actual calls to Isis^2 are missing some important arguments, but don’t worry about that yet; we’ll explain them later. Anyhow, a lookup method is invoked, concurrently, in each of the members. But recall that each member also has information about the current view: the member on the left knows it’s rank to be 0, the member next to it has rank 1, etc. For example, suppose that an airport camera snaps a photo of Mr. Smith and we want our group to rapidly search a database of images with 200,000 documents of known terrorists in it, member 0 could search documents 0 to 49,999, member 1 documents 50,000 to 99,999, etc. You can probably dream up all sorts of clever ways to subdivide a computation. Isis^2 makes it easy to implement them. There isn’t much one can’t do here: everyone has the request itself, everyone knows their own rank and that of the other members too, they agree on how many members were in the group, and can even do things like hashing the request id.

If a member fails, they can see that, and although our example doesn't illustrate the details, they can even program a failure recovery action. For example, someone else could compute part 3 of the subdivided task, since the third member of the group fails in our picture above. Of course, in this example there is no need to recomputed that answer: the member fired off a reply before it died. Isis^2 can even help you know if this is what happened, so that you can program a remedial action only if the reply didn't get sent before the crash.

Notice that this is already an interesting way to exploit parallelism because it can be used in an *online* manner, unlike MapReduce, which is most often used in batch systems that run offline, late at night, and prepare the indicies for tomorrow's web searches and whatnot. Moreover, the query can be delivered in a totally ordered manner, relative to updates. Thus, the state of the data on which the query was executed is guaranteed to be consistent. In contrast, while MapReduce will restart failed parts of Mapped tasks, it offers no guarantees at all about the changing state of any data on which those tasks are running.

One way to take advantage of a group becoming very large is to just issue a request to the whole group, but ask for just *k* members to respond. This isn't a simple matter with Isis^2: *in the current implementation of the Query system call, every member is required to respond to every request* and, if someone neglects to do so, the system automatically sends a NullReply (a real message gets sent even in this case). Thus, even if the query requested just one reply, in the lower levels of the system, N come back. In the future we plan to offer an asynchronous Query in which the Isis^2 point-to-point mechanisms carry the reply back to the query initiator, but this is not yet available. Today, the developer can do this by hand, but it isn't a trivial mechanism to implement.

If a group gets really large, you may want to consider using an *interactive aggregation* pattern, in which a computation is triggered by an initial multicast (just as we saw above), but the result is collected not as a set of n replies from the members to the query source, but instead using a highly parallel, decentralized computation that collects the pieces in a manner very reminiscent of the idea of data reduction. The following picture illustrates the key ideas:

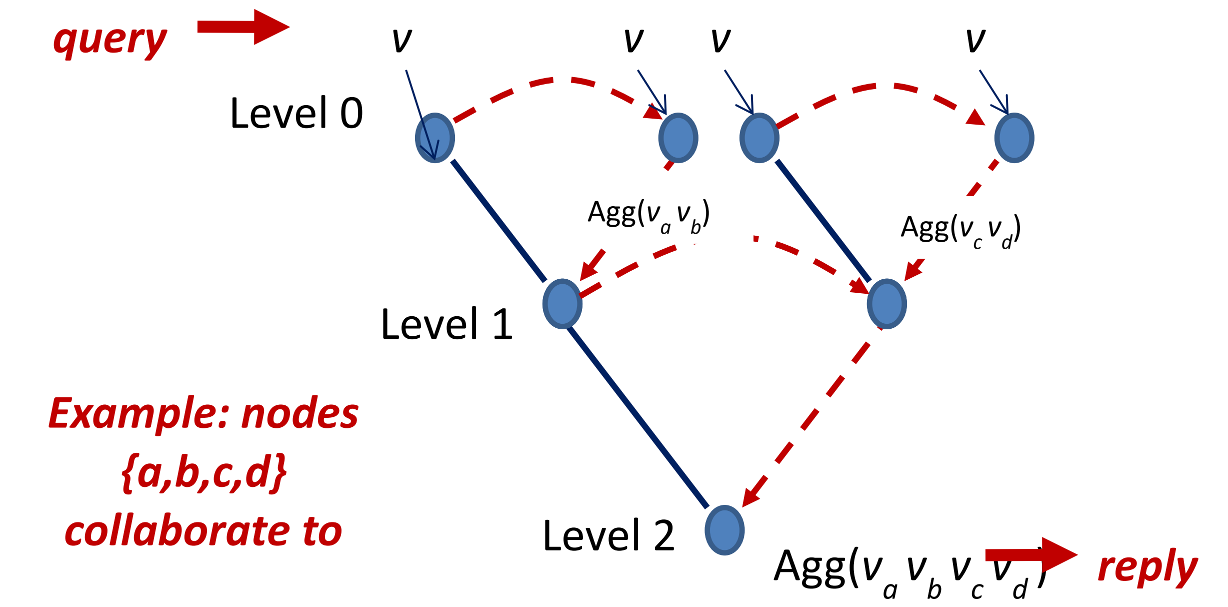


Figure 4: The flow of information in an aggregator.

Interactive aggregation is supported in all Isis^2 groups, but is mostly useful when a group has more than a few tens of members: a size at which many-to-one replies to a query can potentially overload the query initiator and cause high rates of data loss. The scheme works as follows:

1. The group has a leader (currently this is required to be the rank-0 member, and you'll need to monitor the group view to figure out which of your service instances has this special role). The leader has the job of initiating the query and collecting the response: it multicasts the query, or some other event triggers a new aggregation round (we'll discuss a third option, *continuous aggregation,* below). *All the group members will participate in performing the query.* Notice that we are using a multicast Send, not a Query, to perform the aggregation query. Collection of the results is via different mechanism than with a standard Isis Query we saw in Figure 2!
2. Group members receive two kinds of incoming multicasts: those that *update* group state, and those that initiate aggregation queries.
    a. Updates are just applied in the usual way: a handler receives the request and revises the group data structures appropriately.
    b. Aggregation queries are performed, in parallel, by every member in the current view (the view is available, via `myGroup.GetView()`, or by tracking it using a view monitoring method). Thus, every member knows who is in the group (the view "members" property), and every member has its own ranking available (via the `view.GetMyRank()` method). The length of the members list tells you how big the group is at this instant in virtual time; every member sees the same values for the view and hence can use this data to decide who does what.
    c. Having decided which part of the task to perform, each member does its share of the work, and then uses `myGroup.SetAggregatorValue` to tell Isis^2 what it came up with.
    d. If it was expensive computing this result, members might stash a copy of the answer just in case the aggregation "fails" and needs to be restarted, so that the second try will run very fast. (Some extra steps are also needed; we'll explain below).
3. Meanwhile, back at the leader (which is also a participant), a call is issued to `myGroup.GetAggregatorResult`. This method will block until the aggregation has been "swept up" in the manner we'll describe below (the answer is a combination of the *n* sub-answers contributed by the *n* members of the group, and you'll tell Isis^2 how to do the combining).
    a. Normally, the aggregation will succeed and the leader now has the result and can send it off to a remote client.
    b. Sometimes, rarely, an aggregation fails because group membership changed while it was collecting results. If the steps recommended here are taken, this shouldn't happen more than once every ten minutes or so even in a group with 10,000 members. Such a query would normally be reissued; participants might recomputed the request from scratch, but if you've done a good job of stashing recent results, perhaps the result is instantly available and just needs to be resubmitted via `SetAggregatorValue`.

A well-designed aggregation group needs to avoid excessive membership churn; we'll explain how to do this in a section at the end of the manual on dealing with really large systems. By managing planned joins and planned departures so that those occur in batches, you can reduce the frequency of unplanned events (which will all be failures) to a low rate, at which point aggregations would rarely fail.

Aggregation combines nicely with the Isis^2 DHT. To use this combination, you insert DHT tuples using DHTPut, perhaps inserting many at a time, or DHTOrderedPut if you wish to guarantee consistency. Then send your query using the new QueryKey<KT> feature: you construct a QueryKey object by providing an IEnumerable<KT> set of keys that the query will access, and then supply the query key as the first argument to your Send or OrderedSend (immediately after the request code). Isis^2 will deliver the query to a single representative per DHT shard, using the initiator as the rank 0 member. Now, each participant simply computes their portion of the result and uses myGroup.SetAggregatorValue to report it, employing the query key as the identifier for the operation. The initiator waits for the result, again using the query key. This will balance load evenly within shards, and only group members with actual roles to do will participate in the operation!

You can also use a standard Query, OrderedQuery or SafeQuery with an Isis^2 DHT, but in that case all members will receive the request, and the results are sent directly back to the initiator. Additionally, we should note that when ordered multicast is combined with DHTOrderedPut, the system won't take advantage of IP multicast which can slow things down a little bit relative to what you would see in a group that uses ordered multicast but isn't using the DHTOrderedPut operation.

Our running example in this section will be a search of an image repository to look for faces matching an image captured from a camera. For example, one might use such a service in an airport to look for known terrorists or criminals, who might be travelling under assumed names. Let's make a picture of all this. Below, the query comes in at the "top" of this picture, as a multicast to our group members (the illustration just shows four: a, b, c and d). To understand the aggregation step, visualize two rings: one has node a on the left and node b on the right, with node a passing something (call it a "token") to node b, and node b modifying it and passing it back to node a, and so forth. Similarly, node c is passing a token to node d, which changes it and passes it back to node c. Notice that node a actually appears in 3 "levels": level 0, level 1 and level 2, and similarly, node c plays 2 "roles". This figure uses the same arrow of time (top to bottom) as in Figure 2 to illustrate how the aggregation pattern of Figure 3 looks in a timeline fashion.

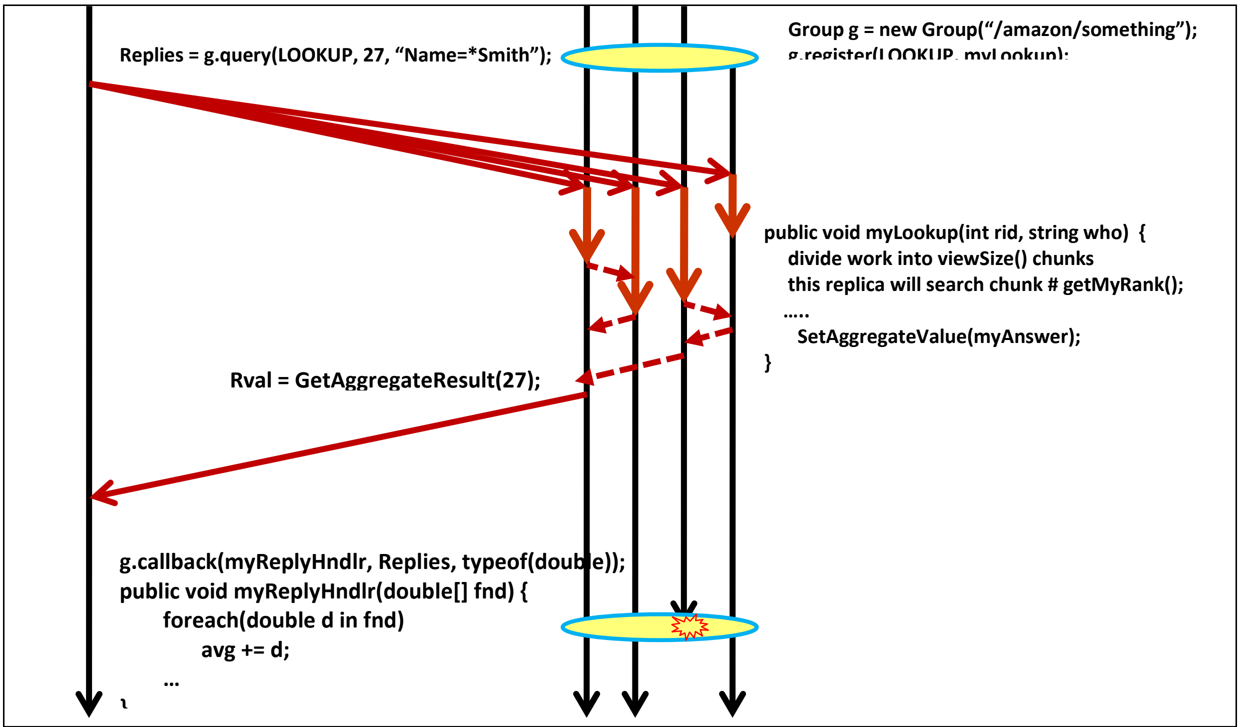
To see how this can be useful, let's compute an aggregated result using the tree of token rings. We'll stick with our image serarch example. The group is managing a database of wanted criminals. The request specifies the lookup to do, and also assigns a request id (19) which needs to be unique (you can design a KeyType of your own, so this shouldn't be hard to arrange).

Accordingly: out goes request 27. On reception, node a checks for matches in photos 0 through 49,999 (we didn't show the group view information, but we still have it: node a is still ranked 0 out of 4, and all that reasoning is still available to us). Perhaps, we find 16 possible matches. Well, the token could carry the count to node b. In our figure node b is running a bit slower, but eventually it finishes computing its own search. Node b only found 3 references, so it adds 3 to 16, and gets 19. Now it passes 19 back to node a.

When the data comes back to node a, and to node c, we treat this as a ring at the next level of the tree: ring 1. At this level, the members of the ring are node a, and node c. So, the results from the (a,b) aggregation are passed to node a, but in some sense a is playing a second role now: as the ring

1 representative for the (a,c) ring at level 0. Similarly for node c. So: node a passes a token to node c saying, in effect, "the (a,b) ring found 19 possible image matches for Mr. Smith." Node c knew about the 21 that (c,d) found, and adds those in. We're up to 40 now, from (a,b,c,d) and this goes back to node a, but now in its level 2 role.

In effect the aggregate is formed along the red dashed lines in Figure 3. The heavy blue lines are just to remind us that the same process is playing roles at more than one level of the tree. Figure 4 shows those same actions but now portrays them as messages passed between the nodes.



```
Group g = new Group("/amazon/something");
g.register(LOOKUP, myLookup);
```

```
Replies = g.query(LOOKUP, 27, "Name=*Smith");
```

```
public void myLookup(int rid, string who) {
    divide work into viewSize() chunks
    this replica will search chunk # getMyRank();
    .....
      SetAggregateValue(myAnswer);
}
```

```
Rval = GetAggregateResult(27);
```

```
g.callback(myReplyHndlr, Replies, typeof(double));
public void myReplyHndlr(double[] fnd) {
       foreach(double d in fnd)
           avg += d;
       ...
}
```

Figure 5: Figure 2, revisited but now with Isis using its Aggregation feature to do the requested computation. To avoid making the figure illegible, the KeyType (int) and ValueType (double) aren't shown here.

Since level 2 is the root of our tree, when the aggregate gets there, node a has the answer to the query. So it replies to the caller: we found a total of 40 references to Mr. Smith. And this was against a well-defined database state (recall the virtual synchrony guarantees), and perhaps even computed with AES 256 security keys on every single message that was exchanged. If a failure interrupts the process, we say that an aggregation is "unsuccessful" and it can be reissued. So any answer is consistent, secure, fault-tolerant. And hugely parallel too. In time proportional to the log of the size of the tree (log in the ring size, actually), our answer pops right out. Our example only had four nodes in the lowest level of the tree, but there could have been *many* more, and if so, all the computation occurs in a single parallel phase. The rest of the work just involves gathering up the answer.

The actual Isis^2 ring size depends on the size of the group. For most groups, the ring is of size 2 and the tree is just a binary tree. The "tokens" are sent on demand in this case: as soon as a value is available (or a pair of values), Isis^2 aggregates the pair and passes the result to the next node in the tree.

But Isis^2 also supports something we call a "large" group, used for really large situations with thousands of members, and for these the ring size is typically about 25. Moreover, here a genuine token-passing mechanism is used, and passing the token is time-based, not data-availability driven So, if our group spanned 10,000 members, and large group mode was activated (as shown below), the request zips to all 10,000 in a single IP multicast. 10,000 members crunch the query, in parallel. Now 400 level 0 rings collect aggregated answers. 25 "ticks" of the clock later we're on level 1, where 16 rings aggregate these intermediary results. 25 ticks later, we have level 2 answers, and 16 ticks later (because our level 2 ring was smaller), we know the result for the whole tree. If we tell Isis^2 to send tokens every 1ms (the fastest rate currently permitted), then in principle, our entire computation takes 0.075 secs and reflects data contributed by 10,000 members. Moreover, there can be many (say, thousands) of queries running concurrently, all being aggregated in parallel, with results pipelined steadily back to the group leader. Of course, getting a less extreme aggregation service to work well would be much easier than pulling off the feat required to actually demonstrate those kinds of ultra-fast numbers: getting a group of 10,000 nodes to achieve this sort of query response and throughput in a steady way would require a great deal of tuning and testing, and the existing testing tools are primitive, to say the least.

Notice that aggregation will be slow compared to directly querying the entire group if the group is small. We don't recommend the use of aggregation if a group will be small enough to just send the replies directly to whichever member issued the Query!

Isis^2 lets you define your own custom aggregators, and they can combine data in any way that makes sense to you. There are some obvious restrictions, of course. One is that an aggregator really shouldn't create arbitrarily long lists. So the "five photos best matched to your sample image", provided that we pass photo id's or URLs and not actual pixels, would be fine. So would mean, median, st. dev, count, and all sorts of other "constant size" (or small, at least) aggregators. But Isis^2 won't prevent you from doing something foolish. You could certainly define an aggregator that would carry larger and larger objects up from the leaves to the roots. Passing those tokens would take longer and longer. And in such cases, Isis^2 will probably malfunction. Performance won't be impressive. A query backlog could form. The system may thrash. Eventually, failures will be detected because this backlog will overwhelm participating processes.

Notation: Declaring a Callback Method "Inline"

Before we say more, it may be useful to just pause for a moment to revisit the in-line coding style convention we mentioned earlier, because we'll use it heavily here. Look at the following code snippet:

```
… stuff …
myGroup.registerAggregator<int, bool>(
(Aggregator<int, bool>) delegate(int key, bool lV, bool dV)
{
        return lV & dV;
});
```

```
… stuff …
```

We're using color-coding to make a point that could otherwise be confusing. In this snippet, the yellow lines are what C# calls an "anonymous" delegate: a method defined in-line that doesn't have a name but is very much like any method you might define in any class of your own. This particular method returns type bool and has three arguments: (int key, bool lV, bool dV). It computes the logical and of the lV and dV arguments, ignores the key, and returns the computed result. The method as a whole has a generic type "`Aggregator<int, bool>`". For those unfamiliar with generics, this is like a macro: a class in which the types "`int`" and "`bool`" will be automatically be substituted for type parameters used in the class definition.

What we see here is that this method is being "registered: with Isis2 via a call to `myGroup.registerAggregator`, which also requires generic type arguments: `<int, bool>`. But the yellow code won't be executed until later. The idea is that first, we register a method that Isis^2 will invoke *later* when it needs to perform an aggregation operation. Thus the code in yellow won't execute until sometime in the future. Meanwhile, the green code will have run long in the past and in some sense, this scope of execution will no longer be active. We'll say more about this coding style later, but you'll see quite a few such examples below.

A person who finds the inline notation very disturbing could also just declare a method :

```
        public bool myAggregator(int key, bool lV, bool dV) { … },
```

and then would register it this way:

```
        myGroup.registerAggregator<int, bool>(myAggregator);
```

We'll stop using the color coding, but the point is the same: you'll call registerAggregator to set things up, and later Isis^2 will call myAggregator to combine the lV and dV values for some key.

The version of registerAggregator seen above is actually a shorthand for

```
        myGroup.registerAggregator<int, bool>(myAggregator, timer);
```

Here, *timer* is a new Timeout(TOms, TOact) object. TOms is a time specified in milliseconds; the default is 15000 (15 seconds), and TOact is an action to take if the timeout expires; currently the only legal value is Timeout.TO_AGGFAILURE, but we may add additional options. The policy is that no part of the aggregation subsystem will wait longer than TOms for a given event. Thus, if some member uploads a value, the system expects it to be "swept up" within TOms; if a member receives a value in the token from the member to the left, at any level of the tree, it expects a matching value (one with the same key) to turn up within TOms, and if the leader has been waiting longer than TOms, an aggregation failure exception is thrown. This prevents the aggregation subsystem from hanging if some member doesn't upload its contribution to the aggregation computation.

We are exploring finer-grained mechanisms that might kill off the offending member, but this involves solving a fairly thorny problem (namely: whose mistake was it?). So, until that problem is fully resolved, we provide just this limited solution. For the time being, the system prints a warning to the console when this happens.

We'll say more about what the leader process should do if an aggregation fails below.

Defining an Aggregator

So, how precisely does one define a custom aggregator? There are three basic steps:

1. Define and register a multicast query handler method that will compute each node's contribution to the aggregated result.
2. Because we'll pipeline queries (many can be active at the same time), define a key that will uniquely identify each pending query. Register the key type (class) and value type if these are not base types Isis^2 would already know about.
3. Define the aggregator method that combines two partially aggregated results and returns a new partial aggregate.

To illustrate these steps, let's define a simple application that might support a door-to-door encyclopedia salesforce. Approaching a home, the salesperson queries the service: is 225 Marshall Blvd in Spokane Kansas a good candidate for a handsome, leather-bound, 21-volume encyclopedia of human knowledge (presumably, up to but not including the Internet era)? Members of the service would now search a massive database of information about buying data, with each member voting "thumbs up" (true) if the candidate looks promising and "thumbs down" (false) if some form of negative data surfaces: poor credit records, or perhaps they bought one last year, or maybe are known to use Wikipedia, own an iPad, or some other evidence suggests that the salesperson would be wasting his or her time. To keep things simple, we want every member of the service to vote, and our goal is to pursue the sale only if all members voted thumbs-up (true).

1. First, the user needs to define a *KeyType* and a *ValueType.* The key type is used to uniquely identify the aggregation instance being performed: any given group might be doing many aggregators of the same "type" at a time, and we need a way to know which value is associated with which instance. The key might be as simple as an integer counter.

   Both types need to be marshallable by Isis^2, and, if you define your own types (classes), registered via Msg.RegisterType. If the KeyType is a user-defined type, the class should also include a "public override int GetHashCode()" instance method. Defining a good HashCode is important; two simultaneously active keys must map to different codes or your aggregator might malfunction.

   You can actually change the hashcode method dynamically in the Isis^2 DHT. This should be done when a view upcall takes place (to ensure that the hashcode will be consistent across the group members). The main use of this feature would be in the context of an algorithm that controls the mapping of keys to individual shards (a so-called "vertex cut" algorithm implemented within an Isis^2 application). Knowing the rule by which Isis^2 maps group members to shards, by controlling the hashcode of the key you effectively tell Isis^2 which shard holds each key-value pair. Should you modify a hashcode in this way, Isis^2 automatically reshuffles the DHT contents to match the new rule. Just call DHTGetHashCodeChanged() to inform Isis^2 that the mapping of DHT keys to shards may be different. We recommend changing the behavior of GetHashCode only on a consistent cut, such as when a new view is reported or upon receipt of an OrderedSend by group members. Obviously, it is very important that GetHashCode behave consistently in all members.

2. Next, you need to design and register an *Aggregation Method* in the group. Given two values it combines them to produce a result value. Isis[2] passes in the key too, just in case it contains information useful to you in computing the aggregate value:

```
myGroup.registerAggregator<int, bool>(
(Aggregator<int, bool>) delegate(int key, bool lV, bool dV)
{
        return lV & dV;
});
```

   Our example uses an int for the key type, and bool for the values; it aggregates by computing the logical and of the values for the whole group. Think of the “down value” or dV as the local node’s vote, and the left value (lV) as the running aggregated answer received from the node to the left on the token ring, and you should be able to match this logic with our figures from above. Note: the aggregator will be called in an Isis thread: you may need to do locking for thread safety if it accesses any form of state from the object in which you declare it.

   In more realistic scenarios, you will often find it useful to encode other kinds of information into the keys and value objects. Keep in mind that any “small” objects will do, and they can include fields that you carry along for the ride but don’t use in comparison. For example, when looking up the city in the United States with the lowest temperature, you would have a ValueType class that encodes a tuple: the temperature recorded, and the city name. The former would be used in the aggregator comparison step, but the city name associated with that lower temperature gets carried along in the value object for delivery to the end-user.

   Below, we’ll see an example that encodes information in the keys too. Keep in mind, though, that the key is specified by the application when it issues the query, so the key can’t encode any kind of “dynamic” information: it will be a constant for a given aggregation computation. In contrast, the value contains anything the local node wishes to contribute.

3. Now your group members can participate in aggregation operations. For example, suppose that our group maintains all sorts of information about candidates for bank loans, and the members will “vote” true or false if they think that an individual is a good candidate.

   Our aggregator will be triggered by a multicast that also specifies the key to use, but any trigger is fine; the trigger could even be elapsed time. The members call a procedure SetAggregatorValue to set the value for a particular key, as in this example, which declares that the calling node has completed its calculation for aggregation myRequestID and the value is true. Recall that our example assumes that the task is to search a database. Suppose the size is NRECORDS, and group member i out of N will search records from NRECORDS*i/N through NRECORDS*(i+1)/N-1.

```
public delegate void GCT(string who, int rqid);

int myRequestID;   // Request id counter used as a key
```

```
myGroup.Send(GOODCANDIDATE, "John Smith”,  myRequestID);
. . .
myGroup.handlers[GOODCANDIDATE] += (GCT) delegate(string who, int rqid)
{
    int myRank = myGroup.GetView().GetMyRank();
    int groupSize = myGroup.GetView().GetSize();
    int from = (NRECORDS*myRank)/groupSize;
    int to = (NRECORDS*(myRank+1))/groupSize-1;
    bool myVote = searchDB(who, from, to);
    myGroup.SetAggregatorValue<int, bool>(rqid, myVote);
}
```

Note: every group member (including the leader itself) should call SetAggregatorValue for any given request. The value contributed using setAggregatorValue, for a particular key, is best thought of as a write-once value that shouldn't change once set (although below, in discussing *continuous aggregation*, we'll see a case that violates this loose rule of thumb). On the other hand, the same key can be reused in different group views, and can even be used in the same view by multiple different aggregators (provided that they have different ValueTypes). Only the rank-zero member of a group can *retrieve* the aggregation value (the group leader). The leader can certainly multicast the value to other group members, but they can't directly monitor the aggregation results.   Here's how the leader (the rank-0 members) collects the answer:

```
bool myRes;
myRes = myGroup.GetAggregatorResult<int, bool>(myRequestID);
```

Note: calls to GetAggregatorResult will block until the result has been computed.   Every aggregated value must be collected; otherwise they accumulate in the group leader, waiting for the leader to call GetAggregatorResult. However, a multithreaded application could issue many requests concurrently, multicasting each query and then calling GetAggregatorResult from separate threads.

Failure handling complicates this very simple model. While aggregation is occurring, failures or joins could occur. When that happens, the aggregation "fails", throwing an exception, which the leader needs to catch, perhaps by assuming a default outcome:

```
bool myRes;
int myRequestID;   // Request id counter
myGroup.Send(GOODCANDIDATE, "John Smith”,  ++myRequestID);
try
{
    // Collect the result for aggregation round specified by myRequestID
     myRes = myGroup.GetAggregatorResult<int, bool>(myRequestID);
}
catch (AggregationFailed)
{
    // View changed before result was known… return a default value
    myRes = false;
}
```

Another option is to loop, reissuing the request and reaggregating the result until successful:

```
bool myRes, success;
do
{
        try
        {
            // Initiate aggregation round myRequestID
            myGroup.Send(GOODCANDIDATE, "John Smith”,  ++myRequestID);
            // Collect the result for aggregation round myRequestID
            myRes = myGroup.GetAggregatorResult<int, bool>(myRequestID);
            success = true;
        }
        catch (AggregationFailed)
        {
            // View changed before result was known… try again
            success = false;
        }
}
while(success == false);
```

When designing code that restarts aggregations in this manner, be aware that partial results will be (automatically) discarded when the view is changed. So each time a request is reissued, participants need to recompute the answer, and even if the request is one that was seen previously, must call SetAggregatorValue again. Indeed, the easiest thing to do is to just reissue the request (as if it was a completely new one), have the participants recomputed their partial results, and then call SetAggregatorValue again. This is what we showed above.

Better Ways of Dealing With Aggregation Failures

Perhaps your aggregation operation was so costly that recomputing answers is very undesirable. In that case, a fancier solution would *reuse the identical request id*. Participants would cache known results for a period of time, and if they already know the result, would re-post it (again, by calling SetAggregatorValue). Isis^{2} only requires that aggregation keys be unique in a single view, so this kind of reuse of a key in two or more rounds of aggregation is legal because they actually occur in distinct views. Notice, however that if the view changed because of a failure, the reissued request might be sent to a group that lacks some of the original members (and of course new ones will have joined too).

Accordingly, when designing an aggregation group that works this way, the group members will need to do a bit of work. Here’s what we recommend. Split your group (somehow, using any scheme you like) into the main workers and the “spares”. Let’s assume that we have S spares in a group of N nodes.

Monitor the group views, and create a List<Address[]> structure to track the “leaving” lists from each view. For simplicity we’ll assume that for view *i* you can just look up the processes that departed relative to view *i-1.* That is, we’ll assume that you don’t garbage collect old entries.

So now suppose that a request was originally issued in view *v* and that it had some sort of request id, which the rank-0 process can generate: *(v:k),* representing view *v*, id *k*. The request throws an AggregationFailed exception, and the leader reissues it in view *v'*.

From our list of departed members, we can easily compute the set of processes that departed in view *v+1*, *v+2,* etc. Let's create a vector by concatenating that data. Now we can map it to our S spare members, perhaps by just having them count off: the first spare takes the role of the first departed member, the second spare the second one, etc. If we have extra spares, fine; if we have more departures than spares, we double up in the obvious way.

So, when the request is reissued, in comes query *(v,k)* in view *v'*. Processes that belonged to the group back in view *v* just repost the answer they computed originally. The spares have real work to do: they compute what deceased process *x* would have computed; what process *y* would have computed, etc. The aggregator works just as if the request was a new one in view *v'*, but now it sweeps up these old results, combines them with the newly computed spare results, and viola: we've recomputed the answer to our *(v,k)* query but done so in view *v'*. You'll also deal with extra spares (have them contribute something that won't change the aggregation answer: a "null" result) and you should keep in mind that if the underlying data set may have changed since view *v*, the spares either should disregard recent updates or might be basing their results on the new data set, rather than the data the original query would have seen. For some applications this won't matter, but if it does matter, you may need a fancier data structure to deal with it, so that a view *v'* computation can somehow run on the data as it looked back in view *v.*

At any rate, with these steps, we avoid recomputing our costly results, and only need to fill in the gaps, which hopefully can be done in a reasonably quick way. The aggregation operation itself is light weight, so rerunning it shouldn't be a big deal.

For example, perhaps request 123 was initially sent out in view 11 of group "Foo". Then Foo had a membership change: in view 11, the members were {a,b,c,d}, but now the members are {a,b,c}. The aggregation fails and is reissued. Some member needs to notice that {d} is no longer in the group, compute that "share" of the result, and contribute it towards the aggregation. This could be done by some existing member (a could do its own share, and d's too), or you could reserve a few spare members for this role. Notice that if a few members have nothing to do, they should still call SetAggregatorValue(some-default-value) to satisfy the requirement that every member contributes toward every aggregation round! (This is what we meant by contributing a null result.)

Continuous Aggregation

Some applications need to continuously track the evolution of an aggregate. There are two basic ways of doing this.

The simplest option is to have some form of counter that advances, perhaps as a function of time. One can then perform an aggregation operation once per unit of time (perhaps, once per second), with the advancing clock generating events that trigger calls to SetAggregatorValue and GetAggregatorResult. Just be careful to do such a call for *all* values of the clock, so that a sudden clock jump forward, or backwards (on computers, clocks can do this) won't fool your application into neglecting to contribute attribute results for some value of the key, which would cause a brief memory leak, and then eventually trigger either complains to the console on the machines involved, or aggregation failures in the leader, or both (this depends on how you set the aggregation timeout).

A second and more subtle option reuses the same aggregator key value again and again. For example, suppose that you always use key 0 in SetAggregatorValue and at some regular frequency do calls to announce new values (that is, every member of the group would implement this behavior). The leader might call GetAggregatorValue for key 0 "continuously" (since the calls block, the frequency will really be a function of how quickly each round completes). In this mode, the single key gets used in more than one "round" within a single view. The leader can track the evolution of the underlying value over time.

One difference to appreciate here, between the first and second of these methods, is that in the former method, each time step of the clock defines a virtual instant in time, reminiscent of our closely synchronous execution in which time steps were shown by vertical dashed lines (Figure 1, bottom left). In effect, the aggregate is a series of snapshots of the state of the underlying system corresponding to the tick of the internal clocks on each node.

In contrast, our second method keeps rewriting the value associated with a given key, say key=0. Here, each time we sweep up an aggregate in the group, some collection of key=0 values are captured and used to compute a value of the aggregator, but meanwhile new key=0 values are bring reintroduced by the group members. Since the movement of the token and the relative execution speeds of group members can vary widely, the aggregate values emerging from the leader might reflect samples from widely different points in time: in place of the regular progression of time one visualizes for the former method, this second method only guarantees that each value used in the aggregate was "more current" than the values used in the prior computation of the aggregate.

Yet for the same reason (the bursty execution speeds of the group members) one could also argue that the smooth advance of time one visualizes for the first continuous method is entirely illusory. In reality, the speed with which time moves will also be jumpy in that approach; indeed, one can argue that in the limit, both methods really give "equivalent" behavior. There is one strong statement the former method allows, however: not only are the contributed values from any given member advancing in time, they are advancing at an average rate of once per second, modulo scheduling imprecision associated with the timer callbacks. The latter scheme will probably aggregate at a much faster rate but it will be less regular. The theoretical argument thus applies only for small timesteps, not for steps that might be as big as 1/second.

Finally, notice that when reusing a key in the manner of the second method, we avoid any risk of memory leaks because the aggregation subsystem only retains only value per member at a time, overwriting it if the SetAggregatorValue routine is called again before the token sweeps up the current value. With multiple keys, as in the first method, the risk of a leak is more real because the system could potentially retain one value per key value, per member, if the application is poorly designed and some members malfunction by failing to contribute values for certain (or for all) keys. However, such bugs will be obvious to the user because they will trigger timeouts (by default, after 15 seconds), and those timeouts will result in console error messages.

Aggregation Tips

Not all aggregators are equally good. Here are a few examples of good and poor ones, for a few example scenarios.

Scenario one: Searching an image database. Let's stick with our example application: it receives some form of input image and should search a massive image database for possible matches. The aggregator's job is to sweep up the results:

Good aggregators:

- The best match found, identified by a tuple giving a score and a URL for the match.
- The best K matches, identified the same way.
- If desired, a short vector "describing" each of the matches.

Mediocre aggregators:

- The best K matching images (you would register a marshaled Image class). This could cause the token to become too large, which makes Isis very slow and can trigger thrashing.
- For each of the best matches, a potentially long vector of match statistics.

Poor aggregators:

- From each member that found one or more matches, a list of URLs for all the pictures that were the best matches. (The issue is that such a list could become extremely long). So: a bounded list is fine, but an unbounded list (or even a very long one) could perform poorly.
- From each member, the actual picture representing the very best match (same issue: the aggregated list will be long and will be passed, hop-by-hop, through many members. Far cheaper is to pass a picture ID number and match quality, but fetch the image afterwards!)

Scenario two: Rapid computation of a financial risk metric. For something completely different, imagine that your application is evaluating the quality of a potential investment and needs a quick risk estimate. The members of the group look at historical trading patterns for the same and similar equities and each member contributes a risk estimator; the combined estimate is a risk indicator.

Good aggregators:

- A running estimate of the quality of the investment on a 0-100 scale.
- The K most serious risk factors to consider, for fixed K

Mediocre aggregators:

- For each of the K worst risk factors, a graphic showing historical data and how this equity fits the picture. Such a graphic could be small, but could have been computed offline; including it in the token causes everyone to compute their contribution, wastefully (since many will be discarded), moreover, the tokens get large.
- A vector of 500 double-precision numbers that represent fits to a risk estimator. (This is constant size, but the size isn't all that small). The Isis[2] design actually allows tokens to become large, but we've tuned the system under the assumption that tokens rarely exceed 16KB in their encoded form. Dramatically larger tokens could stress the system.

Poor aggregators:

- A contribution to the risk picture from each group member (linear in group size)
- A list of "similar" investments based on past performance (potentially unbounded size)

This table may be helpful in designing good aggregation functions. Consider:

Min, Max, Sum	These have constant-size results for the entire group.
Mean, St. Dev	Some statistics can also be computed with a constant amount of space
Best K, worst K	As long as K is fixed and small, the aggregator will be of constant size
URLs, not objects	If the group is managing databases of images or files or web pages, consider giving the end-user a URL for the "best match", not the object itself. The application can fetch the object if desired, and the token will be smaller

To summarize: aggregators don't need to be rigidly fixed in size, but do need to be "small", even when combined over large numbers of group members. It is wiser to pass back a URL than the real object if the object would be larger than the URL. And it would be wiser to have the user fetch the answer in a second stage, via RPC, if the answer retrieves a large object: passing the object along (even if there is only one of them) will mean that the object goes hop by hop through perhaps hundreds of group members. Far better to minimize the data that each member needs to handle, and then let the client application pull the fancy graphic or the real data file in a second step, after it knows which object it wants.

A few additional remarks about aggregation.

Notice that your aggregator method really completes the definition of an aggregator class, consisting of the KeyType and ValueType and the aggregator you provide. Sometimes situations arise in which one wants to instantiate a single aggregation class multiple times, in multiple groups. Isis^2 allows this with one restriction: it uses the KeyType and ValueType as a pair that must uniquely "name" the aggregation operation, within the group. Notice that the aggregator function's type signature isn't considered to be part of the identifier for the aggregator. Thus, you can't use the same KeyType and ValueType *even with different aggregator methods* in any single group.

This shouldn't be too serious a limitation. Suppose, for example, that you wish to define five aggregators that will be used in one group, each using <int,int> as a signature. You aren't permitted to register five different <int,int> aggregators in a single group, but could combine them into a single aggregator, by creating a class of your own (call it KeyTuple) that encodes keys for the five kinds of aggregators. A KeyTuple would thus have a class-id and an aggregation instance id. One can then build a single aggregator method that has five subcases, depending on the value of the class-id in the key (which, as noted above, is passed into it by Isis^2 and hence is easily accessed). Remember to call Msg.registerClass(typeof(KeyTuple)). You will also need to overload the GetHashCode method, for example by computing and returning K_a*37+K_b*27+… (or some other combination of the keys).

Another option would be to just create multiple classes for the different aggregators, each having its own personal KeyType. Each KeyType would just be a class encapsulating an integer. Doing so results in a slightly less efficient packet representation on the wire but the overheads won't be sharply higher, and if this is easier for you or better matches with your team's coding style, there is no reason not to do so.

Finally, as seen in the illustration above, every group member must contribute to every aggregation instance by calling SetAggregatorValue, and every value aggregated must be collected by the leader using GetAggregatorResult. A group member with nothing to contribute should thus call SetAggregatorValue with some form of default or null value. An application that violates this requirement will malfunction: if a member neglects to call SetAggregatorValue, the leader blocks forever waiting for that member to provide its value. If the leader neglects to collect results, and the members (or at least many of them) generate an endless succession of new (key,value) pairs, the system holds them for a while, but eventually the timeout associated with the aggregator kicks in; console error messages complaining are printed in the various members where this happened (so that you can fix the bug), and the aggregation itself may throw an AggregationException.

Large Groups

As mentioned in the prior section, Isis has two implementations of groups: one for smaller groups (which should generally not exceed 250 members; just the same, we've tested with up to 1000 members) and a second for large groups, which can have thousands of members. All the mechanisms discussed up to now work for both kinds of groups.

To put a group into large-group mode, the user calls "myGroup.SetLarge()" prior to the Join(). A large group multicast will always be totally ordered and will not provide the Paxos durability property. The underlying implementation uses IP multicast to send and employs a circulating token ring scheme based on the aggregation logic outlined earlier to implement reliability. The API is unchanged but we recommend that the programmer use OrderedSend() so that readers of the code won't be confused about the properties of the solution. You can also set the configuration parameter ISIS_LARGE to true; this tells the system to use a large group for its own managerial purposes, and will stabilize the platform if the number of applications using Isis^2 has become so large that the core Isis^2 membership tracking service becomes overwhelmed.

Large groups employ a number of optimizations that make sense at huge scale, but would hurt performance in smaller deployments: the time-triggered token passing logic is very slow compared to on-demand transmission of data in the manner used for normal groups. The payoff comes when the on-demand approach might overwhelm a member, much like a distributed denial of service attack: cases where some member asks "everyone" to pitch in, and then gets so many answers so quickly that they literally are dropped because the initiator can't read the data in quickly enough. But to enter this realm of issues, you need really large numbers of group members. For this reason we don't recommend using the large group features of the system unless you expect to be working with thousands of members.

Maximizing Concurrency, Coping with Failures

Notice that in an aggregation group, the leader plays a special role: it sends all the updates, initiates all the queries, participates in every query, and waits for all the aggregation results. If this will overload your leader, consider overlaying two or more aggregation groups on exactly the same members: each can have its own leader, and this will let you perform a kind of concurrent aggregation that spreads the leader work around.

Failures can slow an aggregation system down significantly. This is why we're recommending that you batch group joins (and any planned leaves), and only leave the system to deal with unplanned failures. Those should be rare, but when they occur, the group will hiccup for a period of time determined by the parameter settings you select. Aggregations will need to be reissued and, very often, recomputed (we suggested stashing results, but you may find it tricky to design a workload partitioning rule that works well in that respect).

With these steps, however, it should be possible to keep our 10,000 member group humming. Elsewhere we've investigated the performance of Isis^2 aggregation and demonstrated that it can be configured to maintain a high, very steady, rate of interactive aggregations over long periods of time.

We've used the number 10,000 in this section to convey a subtle message: Isis^2 was designed under the assumption that groups might become large, but that a large group is more likely to have 10,000 members than 100,000 members. Research would need to be done to scale Isis^2 up by a further factor of 10 or 100. We don't know of any reasons this couldn't be done, successfully, but until those steps are taken and experimentally validated, it should be assumed that the system just won't work on larger scales than we've used in our examples!

Comparison with MapReduce

There are several important differences between the Isis^2 style of aggregation and MapReduce. Let's review them here. We noted earlier that with Isis^2, updates to data used in the aggregation are synchronized relative to queries and failures or joins, effectively extending the virtual synchrony or state machine replication model to aggregation. This is a key feature of Isis^2, but not one that will be important for every single application, and even those that need this property can obtain it in several ways.

To understand this remark, recall that Isis^2 supports two ways of querying a group. For a smaller group, one multicasts the query using one of the flavors of Query (Query, OrderedQuery or SafeQuery), then waits for results: perhaps a fixed number, or perhaps ALL. Either way, all the receivers must reply; in the case where only a fixed number are needed, NullReply() messages signal that a particular receiver won't send one of those replies. So: we have a 1-to-all question that elicits an all-to-1 response. This approach is fast, virtual synchronous and guarantees a form of fault-tolerance in the sense that failures have a very well-defined set of possible behaviors. But as groups grow larger, eventually they become unstable: in effect, the members begin to do a "denial of service" attack on the node that issued a query, and it can be overwhelmed. Experiments are underway to determine the maximum size of group that can safely be used this way, but we suspect that the limit will be somewhere in the 100 to 200 range. Beyond that limit the group just becomes too large.

Obviously, for some purposes, one can just split a big group into multiple subgroups and use the subgroups for queries. But in this case updates to the underlying data might no longer be synchronized "across" the collection of groups. As a designer you'll need to think about whether or not this poses a consistency issue in your application.

The other way of querying a group uses our novel scalable aggregation scheme. This can handle larger groups (our testing aims at groups with thousands of members), and obtains stability by avoiding patterns that could overload any member or the sender. On the downside, in large groups the group leader (the rank-0 member) does more work than any other member: all multicasts are actually done by the group leader, and only a thread in the group leader can collect the result of an aggregation. The group itself uses a slower token based means of collecting results of the queries. Although a single group can run multiple queries concurrently the leader will need to allocate one thread per query to collect the results. Thus, it makes sense to imagine the leader handling tens or hundreds of queries concurrently but certainly not more: all those threads consume space and eventually the leader will become bloated, page heavily and slow down until it crashes.

The main advantage of scalable aggregation is that we do end up with a much larger group, and we are given a virtual synchrony consistency guarantee for aggregation over the entire group and it applies even if updates are occurring concurrently with queries. A second potential advantage is that with Isis^2 the application can easily “replan” the mapping step. There are many applications in which intermediary results computed during one step should be used to decide what tasks to perform in the next step. While some of these applications fit MapReduce without too much trouble (one encodes the tasks to be done as intermediary files that become inputs to the next step of the MapReduce computation), there are others in which some code of your own design should run to make these decisions online. Isis^2 easily matches this form of dynamic replanning.

However, MapReduce has features that may be valuable in some settings and that it would be hard (not impossible) to mimic with Isis^2. One is that in MapReduce, the decomposition of a query into tasks is done by the query sender; a task *list* is handed to MapReduce, and it can then allocate tasks from within the list to nodes in any way it chooses, restarting failed tasks if needed. With Isis^2 this mapping occurs either when the query is issued (it could contain a list of tasks, but not an immensely long list), or when the query reaches the group members, and if a failure occurs, the query must be reissued and the aggregation restarted. MapReduce will try to speed up the last few tasks by allocating them to more than one node; nothing similar happens in Isis^2, since there is a one-to-one mapping of subtasks to nodes in our scheme.

A difference concerns multiphase applications. Many MapReduce applications run in stages and leave intermediary data at the nodes in temporary files. One can do the same with Isis^2 but must implement a management policy for those intermediary results, and at present, Isis^2 offers no real help on that problem. We are looking into ways of supporting multiphase applications in the future, but at present, any solution needs to be hand-coded.

Transactional and Non-Transactional Locking

Up to this point we've acted as if every update would be done as a single atomic event requiring a single message. In fact we recommend that if you can get away with that model, you should! In such cases, you won't need global locking: total ordering for the update operations should suffice.

But not everything can fit this particular model: some applications perform transactions, consisting of a series of operations, followed by a commit operation, and in such cases, may need to lock data the operations will touch. If you have such an application, locking is easily accomplished in Isis^2, and this subsection will show you exactly how to do it.

We've implemented a package that carries out this logic in the Isis library itself. The API is as follows:

- g.Lock("LockName"[, timeout]); -- Acquires a lock, returns false if timeout occurs first
- g.WriteLock("LockName, timeout]); -- Acquires a write lock, synonym for g.Lock(…)
- g.ReadLock("LockName, timeout]); -- Acquires a read lock
- g.Unlock("LockName"); -- releases a lock
- g.Holder("LockName"); -- returns the current holder of the lock, else NULLADDRESS
- `g.SetLockPolicies(mode, default-policy, lock-broken-delegate);` -- puts the package into a "locking mode" matched to the intended use case.

  In the event that a read-lock is held by a crashed process, the lock will always be released. However, the handling of write-locks where the holder has failed depends upon the policy. If a write lock breaks because of a failure, an upcall is done to the lock-broken-delegate.

  Default-policy governs the handling of a failure:
  - `LOCK_RELEASE`: If the holder fails, the write lock will be released
  - `LOCK_TRANSFER`: If the holder fails, the write lock will be transferred to the rank-zero group member, and an upcall to the lock-broken-delegate will occur

  Locking modes are:
  - LOCK_INTERNAL: Used only while the group is active
  - LOCK_LOCK_EPHEMERAL_EXTERN: The lock describes external resources but if the entire group were to fail, lock information need not be preserved.
  - LOCK_RECOVER_EXTERN: The lock covers an external object and, if the group fails, when it restarts the lock state should be reloaded from an automatically-created checkpoint.

- `g.SetLockPolicy("LockName", desired-policy [, lock-broken-delegate]);`

The LockBroken delegate has the following signature: `LockBroken(int why, string name, Address holder);` The reason will be LOCK_TRANSFER (to the rank-zero member), or LOCK_BROKEN. The name indicates which lock was impacted, and the holder is the process that failed while holding the lock. Again, note that read locks always break if a failure occurs, and this is a silent action. Only write locks are subject to the policy.

Our package is "non-transactional". Here's how it works, and how one could make it more elaborate.

The package assumes that you have a group to manage your locks. First, you need to decide what the "granularity" of your locks will be. Next, you need to decide how failures should be handled. And finally, you code the handler for lock requests. We'll assume, for the moment, that locking occurs entirely within a group and that the transactions are performed by group members. Later we'll say a few words about the case of external clients that use a locking "service" but don't actually maintain the data in the same group that does the locking (that case is closer to what Google does in their Chubby service).

1. Decide on a lock "naming convention". For example, you might have locks named Var:Inst, such as X:3 or Y:17, using strings to represent these names. There are tradeoffs: locks can be coarser, with broader coverage (e.g. you could have a lock just on "X") or finer grained, with narrower scope ("X:17"). Isis[2] will be doing multicasts when locks are requested or granted, so you should factor in the overheads when designing the service.
2. Decide what should happen if the node holding a lock fails before releasing it. In a transactional application, this may entail discarding partially updated state.
3. Select a request id that isn't in use. You can overload the single request id with more than one handler, so you probably won't need multiple id's for just the single purpose.

The locking tool can be used in two ways. The simplest option is to treat all locks as mutually exclusive, which is how g.Lock() behaves. The slightly fancier option distinguishes read locks (they allow other read locks but not write locks) from write locks (mutually exclusive with all other locks). Either mode, though, implements locking in a *distributed* manner. In particular, a read lock is a global read lock on the whole group – not a local read lock registered purely within the member that requested it. Thus the actual protocol is really identical – all forms of locks are implemented using OrderedSend, as we'll explain further below. The difference is simply that with write locks no other group member can access the locked information at all, until the lock is released, whereas with read locks, other readers are permitted access.

Once a write lock is waiting, new read lock requests will wait until *after* the prior write lock(s) have been granted and then released. This prevents starvation in a case where there is an unending sequence of read lock requests with a few write lock requests interspersed: if we granted the read requests in a way that might let a read bypass a prior write, we could leave the write requests waiting forever.

The easiest way to implement a locking service is to give the group leader a special role of granting lock requests. However, we still need to think about two failure scenarios: one in which the requestor fails, and the other in which the group leader fails. Handling these (both of them) is considerably simplified if the lock requests are multicasts, seen by every member in the group.

It would be tempting, but problematic, to write code like this:

```
If(OrderedQuery(1, LOCKREQ, new Timeout(1000, TO_ABORTREPLY), "X:17", EOL, ok) == 1) {
            . . . do something . . .
            OrderedSend(LOCKRELEASE, "X:17");
}
```

The developer here is probably visualizing a LOCKREQ handler that keeps list of pending lock requests. If “X:17” is already locked when a new request comes in, the handler would need to wait (for example using a Monitor.Wait operation) until the lock is released. But this violates the rule that your Isis^2 shouldn’t “block”; doing so prevents the delivery of additional events from Isis^2 to your application! Accordingly, while the above code has a natural appearance, it gets into trouble when we try and design the implementation of LOCKREQ.

What this adds up to is something we noted earlier: the only practical way to implement blocking in an Isis^2 application is to implement a two-stage operation: the request needs to be decoupled from the completion operation.

One ends up with logic of the following kind:

1. To request a lock, the caller creates a lock id (perhaps, his process address and a counter). Then the caller waits for the lock, perhaps using a Monitor.Wait().
2. The caller sends the request to the group, using OrderedSend.
3. All the group members store the request into an ordered list of pending lock requests.

A release of a lock is done in the same way, with a second OrderedSend. When received, all the members have the same pending request lists, so all can compute the new grant (if some request was pending), or all can mark that the lock is free.

Finally, we deal with failure cases. If a process holding a lock fails, a new View is delivered. If the lock should be released on failure, the View handler for the leader runs, sees that the holder of a lock has crashed, and releases the lock to the next requestor. If the lock can’t be safely released on failure, the lock holder becomes the rank-zero member of the group, and an upcall is done to tell it that it now holds the lock.

What about the transaction that was interrupted?

1. To perform a transaction that accesses X and Y, the application first requests a lock on X, then on Y (we recommend that locks be obtained in *alphabetical order* to avoid risk of deadlock). It multicasts whatever updates should occur. Finally, it multicasts a *COMMIT* message. Group members hold the updates in reserve until the COMMIT, then apply them as a batch.
2. If a process doing a transaction fails, every member can see this via the new View event, and all can discard any partial, pending updates. No COMMIT will occur because once a process has failed, no further messages will be received from it. The locks will automatically be released.

A final question concerns the location of the locked resources. If your group manages locks for “internal” purposes, the lock state can live entirely within the group (LOCK_INTERNAL case, above). Here if the group fails, the lock state reinitializes to empty when the group restarts.

If your locks refer to external resources, you have a choice. LOCK_LOCK_EPHEMERAL_EXTERN mode is used if your lock describes external resources, but even so, if the entire group were to fail, lock information need not be preserved. LOCK_RECOVER_EXTERN: is employed for a lock that lock covers an external object and, if the group fails, when it restarts the lock state should be reloaded from an automatically-created checkpoint. In this case we employ SafeSend to ensure that the lock state will be safely stored on disk before a new lock request is granted.

DHT Support

Modern data centers often use *distributed hash tables* (key-value stores) as a quick way to spread data within a group, storing information into the table by key, and later retrieving it as needed. In Isis^2 this functionality is available for any group that wishes to enable it. The DHT functionality is in addition to anything else your group may be doing, and it works on any kind of group (small or large). Users reading about the DHT may want to read about the .NET Linq functionality, which works extremely well in conjunction with the Isis^2 technology.

The Isis^2 DHT is a "one-hop" DHT: we update it using multicast and we query it using direct point-to-point RPC. This makes it very fast. To activate the DHT mode, call

```
g.DHTEnable(int ReplicationFactor, int ExpectedGroupSize, int MinGroupSize);
```

or

```
g.DHTEnable(int ReplicationFactor, int ExpectedGroupSize, int MinGroupSize, int TTL);
```

This call should be done *before* joining the group. All members must call this method before joining and all must use the identical parameter settings. There are two modes in which DHTEnable operates: in "debug" mode you specify (1,1,1) for the parameters; this lets you test the application using a group of size just 1 or perhaps 2 members. In normal mode you'll use larger values; (3, 6, 6) are the smallest normally accepted by the system. The group itself can be a normal group or a "large" group; the mechanism should work similarly in both cases. In the event of a failure of some member that had key-value data, one of the members at the end of the DHT membership list will be slotted in to replace it, offering a lightning-fast recovery after a crash: this is the only situation in which Isis^2 will report a view that shows members leaving and joining in the same view, and you'll see that the "joining" members haven't really joined; they simply were dropped in to replace failed ones. This minimizes the need to shuffle data within the DHT. Then, with more leisure time, you can launch additional processes to join and replace those that have been lost.

If used, the TTL value tells Isis^2 how long to retain DHT key-value entries. After the specified delay (in milliseconds), a key-value tuple will be automatically deleted. By default this mechanism is disabled and you must explicitly remove items you wish to delete, using the DHTRemove API.

The parameters used in DHTEnable tell Isis^2 how much it should replicate the data stored into the DHT (the target "shard size", that is) and how big you think the group will be during normal executions, and also specify a minimum size below which DHTPut and DHTGet operations will throw IsisDHTException errors (this is because if a group is too small, a key might map to an empty set of nodes). You can catch these errors, and we recommend doing so: after all, churn might drop a group below the minimum size, and perhaps you'll want to code some logic to add more members if that happens. The group would then "recover" from the problem. In our own experiments we often set ReplicationFactor to the square root of the average group size. This seems to be quite a good size for the replication bins in groups that won't have huge numbers of members, but one can operate a DHT safely with any value that makes sense to you: three nodes, or five, for example. Note that you'll need to monitor the group yourself, by watching new View reports, because we don't actively tell you that the size has gone below your minimum unless you call DHTPut or DHTGet.

Once you set your group up and add enough members to it, you can begin to call DHTPut(object key, object value) to put a key,value pair into the DHT, object DHTGet(object key) to retrieve the value associated with a key (you'll get a null object if the key currently has no associated value) and DHTRemove(object key) to remove the value associated with an existing key. Overlays of these operations allow you to insert a list of (key,value) pairs, or to retrieve a list of pairs from a list of keys.

To reiterate a point made earlier in this document, when a collision occurs (same key, but a new value), the default behavior for the system is to discard the value associated with the earlier Put and retain the value from the one done more recently. But you can implement very different behaviors by calling DHTSetPutCollisionResolver() and specifying a method that takes a key and two values (the old and the new), and returns the value you prefer that Isis retain. You could combine the values in some way, discard the old, or even decide between them on the basis of some special approach of your own. Notice that this allows you to create and maintain lists of values associated with a single key. For maximum flexibility, we use type "object" in this API, but keep in mind that you'll typically want a single value "type" for any given key type used in DHTPut. Thus if you do want to maintain a list of values, we suggest that you use List<someType> even in the DHTPut itself. Otherwise, you'll end up with the DHT containing a mixture of key,values that are of form KeyType,ValueType and others of form KeyType,List<ValueType>. You can certainly do this, but it leads to confusing code.

An exciting layer of functionality is available through an API that takes the form g.DHT<KT,VT>(), which is a real-only clone of the locally available key-value pairs that have types matching the given key type (KT) and value type (VT). Use g.DHT() if you just want access to the full local list. This is an IQueryable list in .NET, hence you can apply the Linq operations to the resulting set: Select, Where, GroupBy, etc.

In a typical use case, one would use Send or Query to send some request to the full group membership, and each member would (in parallel) query its local partition of the DHT data. You can then combine the results, deduplicating by calling g.DHTGetPartition() in the members; members that have the same partition-id will have the same DHT tuples. If a DHT has a replication factor of R, you will see R or fewer (because of transient failures) duplicates for each result, but can combine them using Query or using the Isis^2 aggregation mechanism, obtaining a single answer that will robustly reflect the data from the full group.

A variety of guarantees apply depending upon how the API is used. The weakest properties are those associated with DHTPut and DHTGet: DHTPut and Remove run over a special form of FIFO point-to-point protocol, hence it is unsafe to initiate concurrent calls to these functions, *for the same key value*, from multiple group members. With different key values there won't be any issue, but with identical key values, different DHT members could receive updates in different orders.

Substantially stronger guarantees are available if (key,value) pairs are added to the DHT using the DHTOrderedPut operation. This uses a more costly totally ordered multicast to carry out the desired actions for a list of tuples, but guarantees fault-tolerance and totally ordered updates. In this case, by using OrderedSend or OrderedQuery, one can query the DHT across a virtually synchronous *consistent cut*. The built-in DHTOrderedGet works in this same way.

Users fond of MapReduce (Hadoop) can easily implement very sophisticated behaviors over the Isis^2 infrastructure; indeed, far more sophisticated than is common with MapReduce itself. We recommend that you read about the Aggregation features of Isis^2, which can be combined with the DHT layer for purposes such as these.

Earlier we commented that one very advanced feature available to the Isis^{2}user involves dynamically changing the mapping of (key,value) pairs to shards. The idea is this. Isis^2 maps members of a group to shards by using the values you provide in DHTEnable to compute the target number of shards in the group; call this NS. Then from the group view, Isis^2 assigns the member with rank *r* to be a member of shard *r%NS.* In effect, members count off: 0, 1, …, NS-1, 0, 1, … NS-1, etc.

The same idea is used to map a (key,value) pair to a shard: we compute key.GetHashCode()%NS, and this tells us which shard should hold that key. Thus by controlling the value returned by GetHashCode you have complete control over the mapping of keys to shards. Most users ignore this and return a random-looking value that depends on the key but has no special meaning. But if you return a hashcode in the range 0..NS-1 you have certainty that the key will live in the shard you identified.

Moreover, you can modify this mapping. Isis^2 allows you to change the hashcode function for your keys during the new-view upcall event by which we tell you that the group membership has changed. If you do this, when the upcall handler returns, Isis^2 will notice any (key,value) pairs that no longer map to the nodes where they previously lived, and will migrate them to the nodes owning the shard that they now map into. Thus you can dynamically reshuffle the key to shard mapping. Just call DHTGetHashCodeChanged() to inform Isis^2 that the mapping of DHT keys to shards may be different. We recommend changing the behavior of GetHashCode only on a consistent cut, such as when a new view is reported or upon receipt of an OrderedSend by group members. Obviously, it is very important that GetHashCode behave consistently in all members.

Isis^2 also provides features aimed at helping you store complex structures in key-value format. This is where the DHTCollisionResolver mechanism may become especially interesting: in some applications, such as the PageRank algorithm used by Google, one wants to represent a data structure such as a graph. In Isis^2 you would do this using key,value tuples in which the key designates the node (the name of a web page, perhaps) and the values represent edges (links pointing to that node, and the PageRank weight of the page holding the link). With a standard key-value architecture one can only have one value per key, hence you couldn't represent a list of incoming edges. With the DHTCollisionResolver, however, you can code a routine that would form lists or other sorts of data structures that might summarize sets of information in concise ways. One can then start to use a DHT to hold a massive graph and to perform massively parallel computations upon it, such as the PageRank algorithm itself! (We sketched out an implementation of PageRank as an algorithm over the Isis^2 DHT as an illustration of this mechanism in the paper we wrote on about the use of Isis^2 for interactive data analysis).

There are two options for specifying a collision resolver. One is fairly "blunt edged": you provide a method with signature object myMethod(object key, object oldvalue, object incoming) that takes a key-value pair (the key and oldvalue) and a new "incoming" value from a more recent DHTPut and computes a new value to retain (or null if the key-value pair should be garbage collected). The method must be completely deterministic, because it will be executed separately at all the shard members and they would become inconsistent if the same call returned different results at different shard members. You should specify this form of resolver using the non-generic SetDHTPutCollisionResolver() API.

The second option is finer-grained: you specify the key type and value type for the key and old value, and also a second value type for the incoming value, then return a new object with the "old" value type. For example, you could have a key type of string, and an old value type List<double>, and an incoming value of type double; you would merge the incoming value into the list, returning a new List<double>. A null new value can be returned to signal that the key-value pair can be garbage collected. Specify this form of resolver using the generic SetDHTPutCollisionResolver<KT,VT,VTin>(). Note: at present, Isis^2 expects the strongly typed generic resolver to return a result of type VT (for example, VT might be List<T> and VTin could be T). If you wish to have the resolver return a value type other than VT, you should use the object,object,object variant. Thus you could have a resolver that takes two incoming objects of type int and returns a List<int>, but you would specify it using the non-generic object-type registration method, not the strongly typed one.

Finally, we should explain how things behave if you have multiple resolvers that might include mix. The most general case is that we might have multiple DHTPutCollisionResolvers, each with different type signatures, plus one DHTPutCollisionResolver for types all equal to object. Isis^2 will start by checking for an exact match; if it finds one, that resolver will be called. If there was no exact type match, Isis^2 then checks for a non-generic resolver, calling it if one was defined. And last, if no resolver matches, Isis^2 retains the more recent key-value pair and discards whichever value is older.

Thus after a DHTPut is processed, there will be a single retained key-value pair unless you provided a resolver that matched and returned null, in which case garbage collection occurs.

Performance will depend upon several factors. In systems with IP multicast available, DHTPut and DHTRemove (as well as DHTOrderedPut and DHTOrderedRemove) will often be single-IPMC operations, and DHTGet (as well as DHTOrderedGet) will run as a single RPC over UDP. Thus for such settings, extremely high data rates are feasible. In systems that run over pure UDP, performance will be slower for Add and Remove but lookups will still be very fast. In systems that force Isis^2 to tunnel over UDP, these operations will both have delay proportional to the $\log_2$ of the group size, but performance for specific requests can vary because the internal routing overlay has variable length paths from node to node, and the solution makes an effort to find a "short path" among the options available to it.

When new members join a group, Isis^2 figures out which affinity group they belong to and uses state transfer to move the (key,value) pairs they need to the new members, so that they can handle new DHT requests correctly.

Some applications may want to store very large objects into a DHT, or objects that need to persist across failures. For this purpose, you'll want to invoke:

```
    SetDHTPersistenceMethods(DHTPutMethod writerMethod, DHTGetMethod readerMethod);
```

As seen here, you specify a method that will write DHT records (the arguments will be the key and value, and you should return a new "value" that will let you find the data later), read a DHT record (called with the key and the "value" from the writerMethod; it finds the saved data). For example, the DHT writer method could create a file using the key as a file name, the reader could read the file and return the contents. When members join the DHT, state transfer will automatically use the readerMethod to transfer the appropriate objects to the joiner.

When working with a persistent DHT, one complicating factor involves handling of “total” failures, in which all the members of a group crash. The problem is this: because Isis^2 manages the DHT mapping for you, any given (key,value) pair is stored at just a few nodes. Isis^2 routes requests to those nodes, and they handle the corresponding keys. Thus if those nodes are all crashed, DHT requests won’t find the files. Moreover, one must worry about whether the files you create are accessible to new members if the membership of the DHT changes during a crash.

A simple option but a possibly expensive one is to store the files in a global file system and then to use the DHT as a cache that has the role of spreading the work of “owning” those files out in an even way.

Mistakes in the values of the replication factor, the expected group size, or very small values for the replication factor pose a risk. Our random hashing function doesn’t do a perfect job of spreading the group members around in the hash table space: basically, we map keys (and group members) to pseudo-random values, then compute the modulus of those values against ExpectedGroupSize/ReplicationFactor. Thus if you were to tell us that we should do 3-fold replication in a group of 60 members, we end up with 20 bins (as expected).

When you combine the DHT with aggregation, you will need a way to indicate to the OrderedSend or Send multicast procedure the list of members to which a request is targeted. For this, use a `QueryKey`<KT> object. Construct such an object by supplying the list of keys that your query will access. Supply the QueryKey as the first argument to your handler, right after the request code in the OrderedSend or Send. Then use that same QueryKey as the identifier for aggregation operations, if you use Isis^2 aggregation.

One caution: First, the QueryKey object must be the first argument to your request handler, provided immediately after the request code in the Send, OrderedSend or Query/OrderedQuery call. Second, the QueryKey should then be used as the key for the aggregation. And finally, note that *even if the initiator is a member of a shard that none of the keys maps to, Isis^2 will deliver the Send or Query to the initiator as well, so that it can serve as the root of the aggregation tree.* In contrast, without a QueryKey, the initiator would only be included if some key mapped to it.

Oracle Service: What it Does, Where it Runs, How to Override Defaults

Isis^2 employs the Dynamically Reconfigurable Services model of the paper cited in the introduction, and in that model a consensus-based service is required; it decides which processes are healthy and which have failed, and computes new group views and initial states each time membership changes. Our implementation shifts some of this functionality around, but the basis consensus protocol is still needed. The Oracle provides this role, and consists of one or more (normally three) processes that run a state machine replicated membership tracking group.

To simplify life for you, Isis^2 has a built-in implementation of the Oracle within the Isis^2 library that starts itself up automatically in the first three processes that are executed when the system is launched. You don’t need to do anything to get this default behavior; they find one-another using IP multicast beacons, rendezvous, and self-organize. As long as the Oracle processes don't exit, the system will vector Oracle requests through them. If one fails, the two others will add the next process you launch as a replacement. Of course, this may not be your desire: the first three processes you launch might be heavily loaded parts of your application and perhaps you wouldn't want them playing this extra role. The most work gets done by the 0-ranked Oracle member: it runs Dr. Multicast and also functions as the leader for membership and failure detection actions.

For this reason, production users of Isis^2 typically dedicate the Oracle role to specific processes running on specific nodes of their choice. They do this by running the following stub application on those nodes. (Note: The size of the Oracle is currently fixed at 3 using a compile-time constant).

```
using System;
using System.Threading;
using Isis;

namespace Oracle
{
  class Oracle
  {
    static void Main(string[] args)
    {
      IsisSystem.Start();
      while(IsisSystem.IsisActive)
        Thread.Sleep(5000);
    }
  }
```

Normally, Isis^2 applications use IP multicast to find the Oracle. This, however, can be avoided using the ISIS_HOSTS option. Moreover, if Isis^2 is not permitted to use IP multicast (ISIS_UNICAST_ONLY = true), you *must* use ISIS_HOSTS to tell Isis^2 applications where to find the Oracle.

In this case, encode the machine names into the ISIS_HOSTS environment variable, comma-separated: "ISIS_HOSTS=snoopy.cs.cornell.edu,lasi.cs.cornell.edu,biscuit.cs.cornell.edu". Then launch the Oracle on the designated nodes. And finally, launch your application. When ISIS_HOSTS is specified, Oracle functions will be initiated only on the listed hosts.

Help! I've Been Poisoned!

When a problem arises, the Isis^2 system will try to ride out the issue by killing processes that seem to be faulty. It discovers this because those processes aren't responding to ping messages or participating in group membership protocols within a reasonable amount of time. But what value constitutes a "reasonable" responsiveness? After all, as we noted above, only one event can happen in a group at a time; this is part of the virtual synchrony model. Thus, if you receive (say) an update and it takes 1.5secs to apply that update to a disk where you store some large data set, a group join or leave that happens to be concurrent with the update will potentially time out. If this happens, the join logic will tell Isis^2 that your process has failed (it uses the TO_FAILURE case, just as you can use it yourself when setting up an IsisTimeout in a Query).

An application killed by Isis^2 is sent a “poison pill” message. In such situations, the Isis^2 library throws an IsisException(“I have been poisoned”) (there are a few versions with slightly different text strings; they tell the Isis^2 creators precisely what happened, but they all mean the same thing to you as a developer). You may also see an exception indicating that the system “is not running” if you issue a call into Isis^2 concurrently with, or after, a failure. If you wish to do so, you can catch all of these kinds of exceptions, by embedding your Isis^2 logic inside try-catch statements. If so, your application will be able to gracefully shut itself down. However, it is not currently possible to reconnect to Isis^2 after the system has terminated this way. You would need to launch a new process under a new process identifier in order to do so.

If you are encountering inappropriate poison exceptions, the best solution is to consider forking off a new thread to process incoming requests “out of band” so that the threads called from Isis^2 don’t run for very long before they finish. Normally, one does this by connecting the new thread to the Isis^2 upcall via a bounded buffer: a circular list of objects received via the Isis^2 upcall, with a pointer to each end, and methods to Put(object) and Get(object) that block only if the buffer has filled up (for Put) or emptied out (for Get). Isis^2 itself includes an implementation of such a buffer that looks like the code below, using counting semaphores together with a lock; the semaphores make sure that a get waits for the buffer to contain data and that the put waits for it to have an empty slot, and the lock() block protects the corresponding pointers in the event of concurrent get()/put() calls.

```
Semaphore pSema = new Semaphore(BSIZE, BSIZE);   // Has room for BSIZE objects
Semaphore gSema = new Semaphore(0, BSIZE);       // Initially empty
object[] theBuffer = new object[BSIZE];
int pNext = 0, gNext = 0;
                                            // Remove one object from buffer
                                            internal object get()
// Put an object in the buffer              {
internal void put(object o)                     object o;
{                                               gSema.WaitOne();
     pSema.WaitOne();                           lock (theBuffer)
     lock (theBuffer)                           {
         theBuffer[pNext++ % BSIZE] = o;            int idx = (gNext++) % BSIZE;
     gSema.Release(1);                              o = theBuffer[idx];
}                                                   // Next line “helps” for
                                                    // C# garbage collection
                                                    theBuffer[idx] = null;
                                                }
                                                pSema.Release(1);
                                                return o;
                                            }
```

However, this approach works only for slow updates. If the slow operations are queries this won’t solve your problem, since a query must send its reply from the handler procedure, hence if it takes a few seconds to compute the reply, Isis^2 will have timed out and, if you were to pass the query to some other method via the solution shown above, Isis^2 will just generate a NullReply() of its own, which presumably isn’t what you had in mind. In such situations, you might want to consider increasing the default timeout value used inside Isis^2 from the standard 1.5secs to some larger number that better matches the delays associated with your program. Notice, however, that doing so can also make your application slower to detect and react to genuine failures. Moreover, it is important to realize that these timer controlled failure detections are just one of several ways that Isis^2 senses what it believes to be crashes. Thus if you were to set ISIS_DEFAULTTIMEOUT=120000 for example (2 minutes), this won't guarantee that no poison pills are ever sent: the lower level communication layer could still detect outages on the basis of its low-level ping/ack/retransmission logic, which limits the number of times a message will be sent before the communication layer gives up and breaks the connection.

Fast Start

To assist in debugging, you can force Isis^2 to restart the Oracle without first checking to see if the Oracle is already active. This will trim about 45 seconds from the start sequence, but the feature must be used with caution.

To employ fast start, simply have the first instance of your application call IsisSystem.Start(true); The instance will skip the "searching for the Isis Oracle step and will restart within a second or so, as a new Oracle containing one member.

It is important, however, that subsequent instances be launched with IsisSystem.Start(false); or with no argument at all (the default is *fastStart=false*). Otherwise, chaos will ensue.

Using Out-of-Band Data Transfer For Large Objects

A limitation on systems like Isis^2 (also applicable to Paxos, Spread, C-Ensemble, JGroups, etc) is that attempts to multicast large objects can cause serious performance problems and even trigger outright crashes. The crux of the issue is this: in a reliable multicast architecture, everything centers on event ordering: if event *a* is supposed to occur before *b* then anyone waiting for *b* is also waiting "behind" event *a*. Now this may seem like a trivial observation, but suppose that *a* is a multicast containing a gigabyte of data. Event *b* may be delayed for a very long time – so long that timeouts could easily occur. And since timeouts are the basis for failure detection, all sorts of chaos may result: a perfectly healthy application that was doing an innocent operation such as a query could timeout and crash, all because some other piece of code was transferring a big object.

To work around this issue, we recently created a new "out of band" data transfer feature that completely changes the game in settings where the application is working with big objects. The basic idea is to move the big objects in an external channel managed by Isis^2 but outside of the core multicast infrastructure. This way, the very big files aren't "in the data path" used for multicast. In our example, instead of *a* being a multicast or P2P send of a gigabyte of data, we'll just send the name of the out of band object in event *a*. Now *b* won't have long to wait: delivery of *a* will be quick even if the out of band transfer takes a long time. Of course, the application receiving *a* would need to fetch the data associated with *a* since with this change, only the object name is received in message *a* itself. So an addition (potentially blocking) Isis^2 system call is used: the caller asks for the out of band data, and may need to wait if the transfer is still under way. But this solves our core problem.

The Isis^2 out of band mechanism is implemented using what are called *memory mapped files* and more specifically, *non-persistent memory mapped files*, meaning that these files live purely in memory and have no associated disk representation. .NET offers these files through what is called the MemoryMappedFile class: one creates such a file by specifying its name and size, and the MemoryMappedFile handle that .NET provides offers ways to access the associated data segment. In Isis^2 we tend to deal with these files as big arrays of bytes: we call the method MemoryMappedFile.CreateViewAccessor, which returns a kind of "pointer" into the memory region, and then perform bulk Read and Write operations to move data in big chunks. Your code is limited to storing data associated with .NET base types ("non-nullable" data) in these mapped regions: The MemoryMappedViewAccessor API is flexible and permits you to treat a mapped region as if it contained arrays of objects of many different types, but object types are not supported. Accordingly, to map an object, use Msg.ToBArray(obj) to create a byte array representing the object, and to unmap it, use Msg.BArrayToObjects(). This of course will involve copying, hence if you are working with base types such as bytes, doubles, etc, we recommend that you store them as large arrays within your mapped region, not as general objects. You will, however, will need to use the MemoryMappedViewAccessor to access the data… if your program is coded directly in C# and linked with Isis^2. As a result, some users may find it easier to employ the Isis^2 out of band transfer by employing Isis^2 as a "co-process", with some other application coded in a language like C++ creating the objects themselves, and then asking an Isis^2 service to orchestrate replication. With this in mind, we are currently creating an Isis^2 replication daemon that can accept requests via RPC and via a command-line program. You can then store arbitrary data directly in the mapped region, and access it without being subjected to the limitations of the MemoryMappedViewAccessor.

Clearly, there is a form of type safety that can be violated when working in this manner: .NET permits you to create a mapped region containing, for example, 1 million floating point numbers, which Isis^2 will later treat as a byte array containing 8 million bytes. If a mapped object were moved to a machine with a different endian policy or a different floating point representation, you would see this issue directly: whereas data sent through the Isis Msg library would be correctly received, data sent in an object of band mapped file would arrive in the identical format used on the sending machine. But the benefit of this very bare-bones API is that objects are moved with minimum overhead: we transfer them as directly as possible from the memory region on the sender node into the mapped memory regions that we create to hold the data on the receiver nodes.

The API offered is fairly simple ("primitive"). The basic sequence required is as follows:

- Your application first creates the object(s) that will be shared. You can create as many as you like, subject to .NET limits on how many memory mapped files can be open at the same time. Moreover, any group member can create such an object at any time. Use any string that would be a legal file name on all processes in your application for the memory-mapped file name. For example, this code creates a mapped file and fills it with bytes, treating it as a large byte vector (but again, you could use any data type compatible with the ViewAccessor API). In this example, CAPACITY is an integer giving the size of the mapped object and could be very large.

```
MemoryMappedFile mmf = MemoryMappedFile.CreateNew(fname, CAPACITY);
MemoryMappedViewAccessor mva = mmf.CreateViewAccessor();
for (int n = 0; n < CAPACITY; n++)
{
    byte b = (byte)(n & 0xFF);
    mva.Write<byte>(n, ref b);
}
```

- Use g.OOBRegister(string fname, MemoryMappedFile mmf) to inform Isis^2 that you have created a new memory-mapped file with the designated name, and in the memory mapped region corresponding to mmf: g.OOBRegister(fname, mmf).
- Next, you can share the fname within your group. For example, you could use g.Send() to send the fname to group members, or subsets of them. No actual data transfer will occur yet.
- To initiate an out of band transfer, you'll use a method called "OOBReReplicate". OOBReReplicate takes a file name (the file must have previously been registered) and a list of group members, and changes the file replication pattern using out of band data transfer so that each listed member will have a copy, and any non-listed members will not have a copy (if they had one, it will be deleted). Thus each call to OOBReReplicate specifies a new file replication pattern. Data will be transferred reliably and, if your group is secure, will be encrypted prior to transfer and decrypted on arrival. For example, you might issue the following call:

```
g.OOBReReplicate(fname, where, (Action<string, MemoryMappedFile>)
          delegate(string oobfname, MemoryMappedFile m) {
               IsisSystem.WriteLine("ReReplicate finished for " + oobfname);
           });
```

In this example, fname is the file name, and where is a List<Address> giving the group members that should have a replica. The transfer might not start immediately: if there are many concurrent calls to the OOB subsystem, Isis^2 schedules them, performing at most

Isis.OOBMAXIPMCADDRS concurrent transfers within the entire group. The default is 4 but you can change this parameter to any value you like. One IP multicast address will be employed per active transfer, and Isis^2 itself must access each object being transferred, so there are overheads associated with active transfers and very large values of Isis.OOBMAXIPMCADDRS might cause resource exhaustion errors. But we aren't aware of any specific limit that must be respected and a value as large as a few hundred might be possible.

Note: Although a single call to OOBReReplicate can both add new replicas and delete old ones, *the deletion events are performed immediately.* As a consequence of this policy, if you wish to move mapped files outright (e.g. the files were on A and B and now you want to move them to X and Y), you will need to issue two rereplicate requests: one to create the new copies, and then a second call, issued *after the first operation has finished*, to delete the old copies. Although we recommend combining replica creation requests so that a single operation creates as many new replicas as possible, this is not mandatory and in fact more than one OOBReReplicate request can be active on the same file at a time.

- OOBReReplicate may take a while to finish. When done it issues an upcall to the delegate method you provided, if so desired. This lets you keep track of the state of your transfers. You should not initiate more than one OOBReReplicate call on the same file at the same time, but once a transfer completes, you are welcome to call OOBReReplicate again with some other replication pattern. OOBDelete(fname) is a short-hand for OOBReReplicate with an empty replica location list.

  Note: it is *not* safe to just call OOBDelete as the "when done" method in OOBReReplicate. The issue is that OOBReReplicate signals that it is done as soon as the replica is available on each of the remote machines, but this doesn't mean that your remote applications have completed their calls to OOBFetch. Thus if you call OOBDelete instantly, a race condition would arise: the remote members might call OOBFetch "late" and the file won't be available anymore if so. We are considering changing the semantics of the when done method to block until all of those calls to OOBFetch have occurred, but in the current versions of Isis2, there is no protection against such a mistake, so you need to get this right!
- A group member that is "expecting" an incoming file transfer should call OOBFetch(fname). This call will wait until the file has been reliably transferred and then returns the local MemoryMappedFile handle, from which you can create a MemoryMappedViewAccessor with which to access the contents. Here's a simple example:

```
g.Handlers[OOBINCOMING] += (Action<string>)delegate(string fname)
{
    new Thread(delegate()
    {
        MemoryMappedFile xmmf = g.OOBFetch(fname);
        if (xmmf != null)
        {
            MemoryMappedViewAccessor mmva = xmmf.CreateViewAccessor();
            bool good = true;
            for (int n = 0; n < CAPACITY; n++)
            {
                byte b = 0;
                mmva.Read<byte>(n, out b);
                if (b != (byte)(n & 0xFF)
                {
                    good = false;
                    break;
```

```
                            }
                        }
                    }                     }
                    else
                        IsisSystem.WriteLine("File was deleted!");
                }).Start();
            };
```

Some applications may find it useful to specify an OOB Completion Notifier method, using Group.OOBSetCompletionNotifier. If this method is specified, the OOB subsystem will do an upcall to the given method immediately upon successful reception of an incoming OOB file, giving the file name and the MemoryMappedFile handle as arguments. Note that Isis2 currently allows only a system OOB completion notification method to be specified.

This example assumes that g.Send or g.P2PSend was used to notify the recipient of a new incoming mapped file. The receiver doesn’t want to block while holding the exclusive lock on incoming messages, so it forks a thread within which it calls OOBFetch to access the file. Once the file has been fully transferred this thread can advance, and it does so by first checking that the handle is non-null: it will be null if someone called OOBDelete before the transfer occurred. Otherwise, it obtains a MemoryMappedViewAccessor and, in this sample, looks at the mapped file byte by byte to see that it matches the pattern we loaded into the sender’s memory region at the outset (see the code fragment shown earlier).

In this example, “good” will always be true at the end of the check. In fact there is no need to check for reliability; the OOB system is fully reliable.

The pattern of known mapped files and their locations is maintained by the OOB subsystem and can be obtained by calling OOBGetRep. This method returns a List of OOBRepInfo objects, one per object currently registered in the OOB subsystem.

The OOB subsystem offers the fastest possible Isis^2 mechanism for volume data transfer. With a very high speed network device such as Infiniband, the OOB system should be able to keep the network fully loaded and achieve nearly zero-overhead transfers relative to the maximum multicast data rate the network itself can support. However, one current limitation is that we run over the UDP multicast layer, which does not take full advantage of a recent networking feature called the “virtual interface architecture” or VIA. Infiniband “verbs” implements this API, and we are exploring an extension to the OOB technology that would sense the availability of Inifiniband and then use the verbs API to perform direct DMA transfer, memory-to-memory, from the originating machine to the destinations. Thus today’s OOB feature is perhaps 10x or more faster than the fastest Isis^2 multicast option (far more than 10x for really large objects, but we may obtain a further substantial increase in the future.

In very large groups, Isis^2 itself uses the OOB feature to transfer initial views and state when large numbers of members join simultaneously. Thus, if you create a master/worker relationship as discussed in the subsection that immediately follows and then the master adds 5000 members to some group, Isis^2 will form one object for the new group view that each of those members needs, and one for the state transfer to it, and then will employ the OOB transfer system to rapidly transfer that data. Ideally, a single IP multicast will be done for each chunk of the data, dramatically reducing the delay before the full set of 5000 are operational and ready to run your code. In contrast, had the members joined one by one, each membership event would need to be reported to the full 5000, and each new member would need its own private initial view and state. Moreover, for fault-tolerance, each Isis^2 Oracle member normally sends these initial view messages directly to the joining member; when using the OOB technique, the implementation of OOBReReplicate is able to "de-duplicate" and does the transfer just once. If no failure occurs, the OOB infrastructure gives a speedup of literally 1000x or more.

An obvious question to ask is this: if the OOB approach is so desirable, why doesn't Isis^2 simply employ it automatically for large messages of all kinds? Here, there are technical obstacles that arise and understanding them can help you appreciate the challenges for you as a user of the OOB layer. Suppose that some user sends a message containing a 1-Gb object, and that Isis^2, seeing this, were to create a memory-mapped region containing that gigabyte, and then send the temporary name for this region instead of the gigabyte. The message library could then call OOBFetch and recover the data on reception.

But consider the issue of enforcing ordering – suppose that this gigabyte was sent using OrderedSend, for example. It should be easy to see that no message ordered after the OOB one can be delivered until the OOBFetch completes. Thus the OOB transfer would benefit from some speedup (nothing like the 1000x mentioned above; that was for an Oracle-to-many-joiners case, not just 1-1), but the performance hit for other subsystems using Isis^2 remains very high.

Further, imagine that the sender happens to fail so that the multicast gets through, but the OOBFetch will terminate abnormally, returning *null*. In this case, the multicast can never be delivered (because it has a big 1Gb hole in the middle), and yet until it is delivered, subsequent multicasts can't be delivered either. We would have no choice except to terminate the receiver. Worse, because of the way messages are marshaled and demarshalled, if Isis^2 did use the OOB code this way, it would end up doing extra copying on the receive side in order to recreate the message with the 1Gb of data within it. All of these costs and constraints make such a scheme anything but transparent and in fact, explain why we decided not to take such an approach.

In contrast, when the developer takes direct control over the OOB layer, using it under program logic, you can easily avoid such issues. For example, you could create a thread to transfer that very large object and only send the multicast when the transfer finishes; now your receiver code must call OOBFetch explicitly, but on the other hand, none of these other issues arises at all. Isis^2 itself isn't permitted to do the same thing automatically because it would change the sender message order, violating the FIFO rule, but as the programmer you can decide if FIFO sending matters, and in effect can decide when the "send" needs to happen. Thus something Isis^2 can't safely do in an automated way is very much an option for you.

A final consideration involves what we call OOB leakage. Suppose that you initiate an OOB transfer from X to Y but that X crashes as it completes. Y will have a copy of the object in memory, but X may have crashed before telling Y about the pending transfer. Thus Y ends up leaking memory. This is what motivated us to provide the OOB completion notifier upcall: by enabling that feature, Y learns about the incoming object directly, and if the object has some kind of self-explanatory content, Y might even know what to do with it; X wouldn't actually need to send the file name at all in any kind of message. There are other ways to check the contents of the OOB subsystem (via a system call we provide called OOBRepInfo) but this notification scheme is simple and elegant, and fits the spirit of the rest of the Isis^2 system nicely.

We strongly recommend that developers who are working with “big data” scenarios employ these same kinds of ideas to optimize performance of their Isis^2 applications.

Note on mirroring files: It is important to understand that the Isis^2 out of band file transfer layer doesn’t mirror files: it makes copies, based on their contents as of the time that the file replication request occurs. Thus if changes are made to one replica, the other replicas won’t automatically be updated. If you need to update the replicated file and wish to ensure that all replicas remain identical, you need to use g.OrderedSend or some other multicast to send any updates to all the replica holders, which can then apply them in the identical order to their local replicas. In this way, the copies will remain synchronized and consistent.

Building Very Large Scale Systems

Users who wish to build extremely large systems with Isis^2 are faced with several kinds of issues. First, because Cornell lacks testing resources for experimentation on the scale of even tens of thousands of nodes, our knowledge of precisely how well the system will perform at scale is limited. Second, the system itself has overheads that could emerge as scalability barriers, depending on the style of application you hope to construct. You'll need to work in a way that avoids overstressing those aspects of our system. These things said, we really do believe that over time, as we gain experience, it will be possible to build larger and larger Isis^2 applications, with some reasonable expectation that they will perform well even under realistic conditions of the kinds found in the world's large cloud computing data centers.

In contrast, Isis^2 was *not* designed for large-scale WAN network deployments and we do not recommend that the current version be used in such settings. The system would need to be adapted to the environment and if you used the current data-center version in a WAN environment, it would perform poorly and might be quite unstable.

In the remainder of this section, we'll be discussing the key factors that limit system scalability, and our recommendations for addressing them. These are the big issues:

1. The data structures used to track groups and membership are linear in the number of groups and the numbers of members. The Oracle knows about all groups and all membership events, and those events force it to scan the groups list and update membership lists. With very large systems in which such events get frequent, the Oracle will bog down.
2. The core ISISMEMBERS group is configured as a "normal" group by default. You may need to set ISIS_LARGE to true if you see signs that ISISMEMBERS might not be stabilizing (typically, this takes the form of members losing connectivity to the system or complaining that they have been "poisoned").
3. The Oracle also needs to be able to run the Dr. Multicast algorithm periodically, which takes time proportional to the number of groups and, in some respects, to their sizes. This is not an issue in systems where the use of IP multicast is disabled, but Isis^2 performance may be poor in such systems.
4. Individual "small" groups can potentially overwhelm themselves with acknowledgement traffic and other house-keeping traffic, or even with the inflow of replies after a query. This can trigger instability, causing the group members to kill one-another off. The issue would be triggered only as the small group becomes fairly large: not 5 members, but a risk with 250.
5. The very large group structures are a bit slow to instantiate and, once running, can only handle membership churn if you manage membership carefully. On the other hand, these are aimed at groups that could have tens of thousands of members.
6. Precisely because of consideration (4), very large groups implementing the Isis^2 aggregation mechanism may fail to make progress if aggregation isn't designed carefully.

How can you help make life easier for Isis^2? Let's start by focusing on the Oracle.

As you know, small deployments of the system can be relatively nonchalant about the Oracle: the first three nodes launched will take on the Oracle role, and because the application is small, the work associated with doing so will be modest, even if you were to launch a few dozen application processes simultaneously.

With larger configurations, however, we assume that you will run a small number of dedicated Oracle processes on a small set of machines (we recommend 3), specifying these in ISIS_HOSTS. On a well provisioned machine, the Oracle should be able to handle thousands of join and leave events per second. The Oracle members just call IsisSystem.Start() and then can call IsisSystem.WaitForever(); this call will block until something causes the Oracle node to terminate, at which point it will throw an IsisException(). You can also loop, calling Thread.Sleep(), then checking the bool variable IsisSystem.IsisActive.

To reduce the workload associated with join and leave events, it is helpful to batch the joins and, if processes will leave groups voluntarily, to batch the leave events or "terminate" the group. These actions are typically done by designating some node as the group master; it will control membership in the group or groups. Here's an approach that works for us. The Master first registers a callback routine via IsisSystem.RegisterAsMaster((NewWorker) delegate(Address theWorker) { … })). Then it launches the worker processes, giving its own Address as an argument (to make this easy, you can call IsisSystem.GetMyAddress().ToStringVerboseFormat() in the master, which returns a long-format printable string version of the master's Isis Address. This can be passed to the workers, and then they can call new IsisSystem.RunAsWorker(String myMaster) to tell Isis that they are workers of this particular master. At this point the master calls IsisSystem.Start(), then waits, collecting the Addresses of its workers via callbacks to the delegate method specified in RegisterAsMaster(). Finally, when it has the full set, the master calls BatchStart() and the delegate begins to call RejectWorker() for any additional workers that turn up "too late".

In situations where the workers know the IP address on which the master is running, you can also call IsisSystem.RunAsWorker(String myMastersIPAddress) in the worker. The IP address should be in the standard IPv6 string format (for example, "123.45.67.890").

As a side comment, in situations where you don't know the IP address in advance, passing the master's address to the workers can be challenging. We usually do this by creating a shared file, but keep in mind that shared file systems can be slow to propagate updates. A method that worked on our cluster was this: initially delete the file (say, MasterAddr.txt). Have your script call sync() and then delay a few seconds, perhaps 5. Then create a new one in your master containing its address in verbose form. In the workers, loop trying to read this file, then sleeping 1 second or so. The bug to be wary of would be a situation in which the workers manage to read the OLD file (cached in the file system) rather than the new one, and then try to connect to a previous instance of your master Isis program. If this happens, you need to somehow flush the old stale cached file out of the worker file system instances. With a bit of hacking you should be able to manage this by adept calls to sync() and sleeping a few seconds at the right place. Sorry this is so clumsy, but somehow, you do need to get that master's correct address to the workers!

Next challenge: We want to make sure workers get a chance to set up handlers for any groups they might be added to in a batch style. In the example below, we let the master pause to wait until the workers are synchronized. As you can see, the code uses "WaitForWorkerSetup". The idea is that the workers initially were blocked in IsisSystem.Start() until the batch start, but then start running. We give them a chance set up their groups (calling new, attaching callback handlers) – except that they don't ***do the join themselves***. Instead, they finish setup and call WorkerSetupDone. At that point, the master can do a MultiJoin to add the workers all at once to the group we're building. Meanwhile the workers themselves should wait until the groups have their first view.

Here's an example:

```
    static void beMaster(string[] args)
    {
        bool done = false;
        List<Address> myWorkers = new List<Address>();
        IsisSystem.RegisterAsMaster((NewWorker)delegate(Address hisAddress)
        {
            lock (myWorkers)
            {
                if (done)
                    IsisSystem.RejectWorker(hisAddress);
                else
                    myWorkers.Add(hisAddress);
            }
        });
        IsisSystem.Start();
        string myAddr = IsisSystem.GetMyAddress().ToStringVerboseFormat();
        int MYGOAL = 1000;
        // Platform dependent: launch MYGOAL (or more) copies of your program on
        // some set of hosts, pass myAddr as an argument.
        // It will be a string that looks something like this:
        //               (6751:126.77.51.13/1556/1557)
        // Next wait for them to connect back, which shouldn't take very long
        int count = 0;
        while (count++ < 60)     // Stop after 60 seconds or when enough are up
        {
            lock (myWorkers)
                if (myWorkers.Count() == MYGOAL)
                {
                    done = true;
                    break;
                }
            Thread.Sleep(1000);   // 1 second delay
        }

                               Continued on next page


        // If our start sequence failed, abort.  Else we're in business!
        lock (myWorkers)
        {
            if (myWorkers.Count() < MYGOAL)
                throw new Exception("Unable to launch enough copies!");
        }

        // Now we activate all the Workers simultaneously.
        // (They will block on Isis.Start() until this line executes in the master)
        IsisSystem.BatchStart(myWorkers);

        // This delays until they have all finished their batch start
        IsisSystem.WaitForWorkerSetup(myWorkers);
        Group.MultiJoin(myWorkers, new Group[] { myGroup });

        // In front of this next line do whatever you want this application to do
        IsisSystem.WaitForever();
        // If the master shuts down, its workers will too
        IsisSystem.Shutdown();
    }
```

What does a worker do? Rather than doing the normal IsisSystem.Start() sequence, the worker processes need to first learn the Address of the master (normally as a command-line string argument. Then they call the procedure IsisSystem.RunAsworker(myMaster), passing in the Address so obtained. This will block until the master calls IsisSystem.BatchStart(). A worker then does anything a normal Isis process can do. Exceptions are thrown in the worker if the master rejects this process as a worker (via RejectWorker()) or if the master terminates.

A worker who lets the master do a multi-join to add it to the group rather than doing the group join itself (which can overload the system just like a parallel start would have) might look like this:

```
    static void beWorker(string[] args)
    {
        // This next line assumes that argument 0 is the master's Address
        // You can also use RunAsWorker(new Address(mastersHost, 0))
        // if you know the host IP address of the master but not the pid.
        IsisSystem.RunAsWorker(args[0]);

        // This line blocks until the master issues the BatchStart() call
        IsisSystem.Start();

        // Before calling this next line do whatever setup this worker must do:
        // create your group handles and register callbacks – but don’t call Join
        // For example, you might call g = new Group(“something”), then call
        // g.ViewHandlers += myViewHandler; … etc – anything needed to have the
        // group ready for a Join.  But you call SetUp done INSTEAD of g.Join().
        IsisSystem.WorkerSetupDone();

        // Now, for each group the Master created using a multijoin, you wait
        // for its first view to be reported.  This is one way to do that:
        foreach (Group g in myGroups)
           while (!g.HasFirstView)
               Thread.Sleep(250);


        // WaitForever would freeze the main thread but if the worker has joined
        // groups (or gets added to groups by the master using MultiJoin(), the
        // worker could be quite active, receiving messages, sending them, etc)
        IsisSystem.WaitForever();

        // If the master shuts down the worker will throw an
        //                     IsisException("master termination");
        // If this next line actually executes, this particular worker will exit
        // (in effect, this worker is a normal Isis application by now, except that
        // if the master terminates, it does too.  In particular, it can
        // deliberately chose to leave the system if it wishes to do so
        IsisSystem.Shutdown();
    }
```

Once the workers have started, the master should use MultiJoin() or MultiCreate(), adding the workers to multiple groups, if desired, and adding many at a time (perhaps, thousands or tens of thousands). To reduce the frequency of node failures, be careful to use the MultiLeave() system call or MultiTerminate() calls, so that groups can be reduced in size or eliminated using just a single operation. Such steps are hugely beneficial for performance.

Notice that there is a slight race condition here. After the master calls BatchStart(), it shouldn't call MultiJoin() or other multigroup operations unless the workers have set up the relevant groups and registered event handlers for those groups (in effect, done everything except to call group.Join() or group.Create()). But how can the master know that the workers have reached that point?

For this purpose, the leader calls BarrierWait(List<Address> myWorkers); this will delay until the designated processes have all called BarrierReached(). Thus, the workers can call Isis.Start(), then create new groups and register the handlers needed, and the leader won't try to activate those groups until the members are ready for traffic in them.

A single Oracle can handle thousands or tens of thousands of groups and can keep up with many hundreds of join/leave events per second… but not more. If you will use Isis^2 on a larger scale, break your application into multiple disjoint subsets, each using a different Oracle. If you expect churn to be frequent take actions to batch joins, leaves, and other events.

Even with these steps, heavily loaded small groups may be unstable above a size that will depend on the speed of your network and your nodes, but will probably turn out to be between 250 and 500 members. Larger groups would need to use the LargeGroup APIs. With smaller groups, the pattern of communication will have a significant impact on peak performance. For example, small groups can handle high rates of concurrent Send() operations, but because OrderedSend() needs to place concurrent events into a total order, performance will be lower and the instability point reached sooner if you use OrderedSend() from many sources. Designs that do only Send() and issue them from a single source will often scale far better; this is so pronounced an effect that you might want to consider creating separate but perfectly overlapped groups, one for each Send() source!

Let's focus now on large groups. Here, the first thing to keep in mind is that in very large groups, all multicast events (updates and queries) actually route through the rank-0 member. This obviously limits the possible multicast and query rate. Indeed, queries become problematic in large groups: clients of the large group can query their local representative in a load-balanced manner, but true Query() calls done by group members can become overwhelmed by the convergence of replies (even NullReplies), causing a kind of DDoS attack on the rank-0 member!

Accordingly, for extremely large groups, we recommend a design that multicasts updates (but batches small events and in this way, sends fewer updates, aiming for somewhat bigger messages sent more rarely). Vast numbers of clients can be load-balanced over the members via a standard web services structure or using the Client() API, if the clients are themselves Isis^2 applications. It may also make sense to create some small number of exactly superimposed large groups, each with a different rank-0 member. This allows some degree of load-balancing, but at a price: the different groups will each have FIFO ordered multicasts from different senders, hence updates in the different groups can arrive in different relative orders. Thus to adopt this approach, you need to think of each of the groups as "owning" a distinct subset of the group data, with any given group holding an update lock on its own subset of the data. Queries can then access data across all the superimposed groups, and data will be consistent, reflecting all the updates up to some point in each group.

We're not suggesting that you create a huge number of these superimposed large groups. In our experiments and testing we've worked with ten superimposed groups, so you will be breaking into uncharted waters if you were to try this with hundreds. (If you do, let us know how that goes.)

Aggregation queries can run into scalability issues as well. First, as a group gets larger, the time needed to collect the result of an aggregation rises, roughly logarithmically in the size of the tree. One must visualize a kind of pipeline of pending aggregations, converging towards the rank-0 member. To achieve good scalability, this pipeline cannot become overloaded; otherwise, nodes will begin to fail, reporting that they have been "poisoned". This means simply that the node fell so far behind that other processes in the group mistakenly believed it was dead.

Assuming that the aggregation pipeline doesn't get overloaded, the next worry is that membership churn (rapid joins and leaves) could cause aggregation to fail frequently. Batching joins and planned departures can help: now we only need to think hard about outright failures, and one hopes that those will be rare.

With large enough groups (a friend of ours at Microsoft asked, on reading this manual for the first time, how Isis^2 would deal with 400,000 member groups[5]), this phenomenon could be a real problem. Assuming nodes crash once every 20 days on average (a common industry figure), a 400K-node large group will experience one true failure every 20 seconds. Thus one would need to parameterize Isis^2 to "handle" these events within a few milliseconds, or the group will have very frequent hiccups, each causing large numbers of aggregations to fail.

The parameters in question related to the way that Isis^2 senses crashes and the rate at which tokens circulate: both will need to be pushed to very low values, which trades higher background overheads for lower delays in detecting and reacting to failures. The layout of nodes could become important: one starts to want nodes to be near their neighbors, not on the opposite side of the data center, because we want low latencies and local message passing overheads.

Very large configurations also demand sensible aggregator designs. If an aggregation on 400,000 nodes fails, we might not want the same work performed again and again. Instead, group members should cache recent results so that if the same request is reissued, members can potentially retrieve an old result and report it again. But this, in turn, suggests a style of workload decomposition that won't be greatly impacted by membership changes.

[5] We told her that we have no idea: as good software engineers, we don't believe a thing works until we've done serious stress testing. The current version of Isis^2 was tested in scenarios that give us some confidence that the system is robust with as many as 10,000 processes using it. But we simply have no experience and hence no confidence that it could run with 400,000!

All of these considerations add up to a fascinating research topic. We would love to work with you to tackle a question of this kind, if you'll let us publish on the resulting improvements to the system. But you are very unlikely to get Isis^2 to work well on this scale without our hands-on help!

User-Controllable Runtime Parameters

Start(noConsole, fast)	There are two overloads to IsisSystem.Start(). One allows you to start "fast": if this flag is true, the instance you launch will skip the step of looking for an existing Isis system and just assume that Isis is restarting from scratch. If you plan to launch N processes you clearly could only use this feature in 1 of them; the processes won't find one-another otherwise. The second overload has an additional parameter: noConsole. If this is set to true, any output generated by the Isis library will be directed to the System.Diagnostics.Debug stream rather than to the Console stream. Note also that there is an environment variable, ISIS_MUTE, with which you can disable all output from the Isis library, entirely.

Isis provides some user control via per-group methods. These include the following:

g.SetLarge()	Tells Isis to use the large-group algorithms for group g. Set ISIS_LARGE to true if you want Isis^2 to call SetLarge for the ISISMEMBERS group, which is used to track the health of system members.
g.SetSafeSendThreshold()	Provides a value Φ that will be used when SafeSend is issued to the group. As you read earlier, SafeSend is like the famous Paxos protocol, and guarantees that multicasts will be durable and ordered. But how many copies need to be delivered to consider the message durable? By default, we wait until *every* member of the group has a copy, but in a group that could have 100 members (or 100,000) this could be a very long wait. If you specify a value for Φ, we wait until Φ copies have been acknowledged and then allow delivery upcalls to occur in all members. For Paxos, this corresponds to the number of acceptors that must acknowledge a message, while the full group defines the learners (that is, Φ members play dual roles, and all *n* members in the current view play the learner role). The acceptors are selected as the low-ranked Φ members of the group, and the sender is counted as an acceptor, hence we actually wait for Φ-1 acknowledgements from members other than the sender.
g.SetDurabilityMethod()	Allows the user to substitute a new definition for "durability". By default, SafeSend considers a message to be durable when an *in-memory copy* exists at the number of members specified by the SafeSendThreshold parameter Φ. With this call, one can replace the standard durability method with some other method. The argument to SetDurabilityMethod is an object implementing the IDurability class. One prebuilt class of this kind is available: we call it the DurabilityLogger and if you instantiate it, our code will automatically store pending SafeSend multicasts in the checkpoint file of a group. On recovery from a total failure, these pending multicasts will be completed automatically. One obtains an at-least-most behavior: the messages will

	be delivered but perhaps, rarely, twice (e.g. if an update was delivered just as the power to the whole data center failed: some might receive and apply it and others not; on recovery, the DurabilityLogger would redeliver such multicasts, leaving it to the application to discover and reject duplicates). For example, the call: g.SetDurabilityMethod(new DiskLogger(g, "myLogFile")) would configure SafeSend for group g to maintain disk logs of the messages being sent in log files with the name "myLogFile:n", where n identifies the acceptor responsible for the particular log file. You can also substitute your own durability logger methods; see the online documentation for g.SetDurabilityMethod for details.
g.UseUnicast()	Tells Isis to only use point-to-point UDP messaging for this group
g.UseIPMC()	Asks Isis to try to assign this group an IPMC address. This request will not override the ISIS_MAXIPMCADDRS limit as configured into the Oracle.
g.TraceMsgs(OnOff)	Prints a trace to the console and to a log file of messages delivered in group g, including low-level system-generated messages.

In Linux or Windows one can easily set environment variables using shell commands or at runtime, if you prefer to do so.

For example you could issue the bash "export" command: "export ISIS_HOSTS=192.67.41.10".

Isis^2 scans the environment variables for the following runtime parameters:

Configuration Parameters Commonly Modified by the end user

ISIS_PORTNO	Base of a range of port numbers used for IPMC to Isis Groups. *If a firewall is present, the firewall policy must not "block" communication to this port.*
ISIS_PORTNOp	A port number used for Isis "point to point" UDP traffic. Isis also uses ISIS_PORTNOp+1 for acknowledgements, and if doing out of band file transfers or membership view transfers over TCP, it uses ISIS_PORTNOp+2 for the TCP connections. Note: if several copies of Isis are launched on a single machine, so that all share the same single IP address, the first launched will use (ISIS_PORTNOp) but subsequent copies will use other port numbers. *If a firewall is present, the firewall policy must not "block" communication between these blocks of ports or the system will not start properly.* **On Linux, the firewall is called "iptables". On Windows, the "Windows Firewall" controls port blocking. You must set these properly for Isis^{2-} to tunnel through them.**
ISIS_HOSTS	In ISIS_UNICAST_ONLY mode, lists the machines where the Oracle will run
ISIS_UNICAST_ONLY	Forces Isis to not use IPMC at all. This is often used on EC2 because on EC2, hardware IPMC isn't supported. However, EC2 does have a kernel network driver that will emulate IPMC in software, hence in

	fact Isis^2 can be used with its default settings on that platform. Today, many of our EC2 users run without setting ISIS_UNICAST_ONLY=true. We haven't done the detailed performance tests needed to know which works best.
ISIS_MCRANGE_LOW	Low end of the IPMC address range Isis should use
ISIS_MCRANGE_HIGH	High end of the IPMC address range range
ISIS_MAXIPMCADDRS	Limits how many IPMC addresses Isis will use at a time
ISIS_SIGS	Enabled (true) by default; tells Isis to protect itself against corrupted or non-Isis messages by including SHA2-256 signature on each packet and confirming that each received packet has an appropriate signature before processing it. Can be disabled for extra speed, but we recommend against doing so.
ISIS_AESKEY	If desired, Isis can encrypt the SHA2-256 signature. You will need to generate a 256-bit (hence, 32-byte) AES key, providing it to Isis^2 as a string through the ISIS_AESKEY argument. Isis will scan this string two characters at a time, interpreting each pair of characters as a hex representation of the corresponding byte of the key. The system will then enable ISIS_SIGS and will use the key to sign each SHA2-256 hash, then verify the signature on incoming messages. Provided that you maintain the key in a secure way, intruders will be unable to generate poison pills that Isis^2 would accept.
ISIS_DEFAULTTIMEOUT	Delay until timeouts in protocols such as the ones used when a join or leave causes group membership changes.
ISIS_SKIP_FIRSTINTERFACE	D IP multicast on the first network interface. Needed on Emulab where the first network interface is reserved for control traffic.
ISIS_NETWORK_INTERFACES	Lists the network interfaces Isis can use. These names should be strings from the names printed by ifconfig –a (or /all). The system will limit itself to these interfaces and won't send packets on those not in the list. Normally, Isis2 binds to every interface.
ISIS_NETMASK	Provides a mask that will be used to select network interfaces on which Isis^2 can send and receive multicast messages: the system masks each interface address with ISIS_NETMASK and checks for an exact match of the resulting masked bits against ISIS_SUBNET. 0 by default.
ISIS_SUBNET	If specified, used to select network interfaces on which Isis^2 can send and receive multicast messages: the system masks each interface address with ISIS_NETMASK and checks for an exact match of the resulting masked bits against ISIS_SUBNET. 0 by default.
ISIS_LOGGED	Enables logging of Console messages to a log file. True by default in the beta release of Isis2, but you can override the value.
ISIS_LOGDIR	Gives a directory in which log files can be stored. Default: "logs" in the directory where the application finds itself running.
ISIS_USEIPv4	True by default. Enables use of IPv4 addressing.
ISIS_USEIPv6	False by default. Enables use of IPv6 address (*not yet supported*)
ISIS_GRACEFULSHUTDOWN	Asks Isis to attempt a graceful shutdown if IsisSystem.Shutdown() is called. By default, the system shuts down by calling the .NET Environment.Exit operation, which terminates the entire process.

Configuration parameters that would normally ***not*** be modified by the end user

ISIS_RATELIM	Normally 0 but can be useful when sending RawSend messages at high rates. Sets a limit in packets/second (UDP packets) on the rate that Isis can transmit. For example, if RATELIM=2500, the system won't send more than 2500 packets per second on average over any 1-second period. The issue here is that with RawSend, flow control and reliability mechanisms are disabled, hence you can easily send so rapidly that you'll overflow the receive sockets on the destination machines (there needs to be enough space in the kernel to hold as many packets as might arrive before the O/S schedules the receive thread, and this is surprisingly easy to trigger). In our experiments with RawSend we usually set ISIS_RATELIM=2000, but otherwise we leave it at 0, which permits infinite rates. Flow control does perfectly well in "normal" situations.
ISIS_OOBVIATCP	True by default. Tells the system that if running with ISIS_UNICASTONLY=true, as on EC2 for example (EC2 is very limited this way), to use TCP to do out of band file transfers rather than the native Isis^2 UDP-based point-to-point logic. This will give a speedup of about 20% on such platforms.
ISIS_USERDMA	Tells Isis that it should attempt to use remote DMA transfers in the out of band file transfer logic. This speeds things up but is only possible if you have installed the RDMA package on your Windows or Linux machine. If you set ISIS_USERDMA=true but have not installed the Verbs software, the system won't start up. With ISIS_USERDMA you will also need to link Isis to the library ib.dll, which is available in our distribution. If you don't download ib.dll and ISIS_USERDMA is false, ib.dll is never used and hence the system won't notice if it is missing.
ISIS_INFINIBAND	If set to true, tells Isis that you have configured it to run over an Infiniband network and that it should optimize its RDMA transfers to use Infiniband's special high speed features. This makes out of band transfers of big files very fast. Ignored if ISIS_USERDMA=false. If you have multiple network interface cards and want to use this feature you must also use ISIS_NETWORK_INTERFACES to tell the system which interface it should use. ISIS_INFINIBAND automatically puts the system into ISIS_UNICASTONLY mode. As a result you must specify a value for ISIS_HOSTS so that the system can start up properly. In practice it will bootstrap itself using UDP communication and then switch to using Infiniband after the bootstrap step completes. Note that ISIS_INFINIBAND uses Infiniband for all messages, not just out of band ones. The reason we mention that it speeds up out of band transfers is that normal messages are small and heavily encoded and may even be encrypted. Thus the benefits of ISIS_INFINIBAND are smaller. But out of band files can be very long and are transferred in a single massive copy (and we have an even better solution in the works). This allows you to take full benefit of the Infiniband network device. The speed can be 100x or even 1000x faster than what Isis itself would normally achieve!

ISIS_FASTETHERNET	If set to true, tells Isis that you have an Ethernet device that runs at high speed and supports the RDMA over Ethernet (ROCE) feature. In this case Isis will optimize its RDMA transfers to use the RMA feature of your fast Ethernet card. This makes out of band transfers of big files very fast. But not every Ethernet vendor supports RDMA Verbs for Ethernet. Ignored if ISIS_USERDMA=false.
ISIS_LARGE	Tells Isis that the application has so many members that the core ISISMEMBERS group used by the system itself should be configured as a "large" group. In fact you can certainly set this to true if needed, but you shouldn't find that necessary unless your application has thousands (perhaps tens of thousands) of members.
ISIS_MUTE	Tells Isis not to print debug messages to the console. You can also get this effect by setting the noConsoleMsgs flag when calling IsisSystem.Start()
ISIS_IGNOREPARTITIONS	Tells Isis not to worry about apparent partitioning failures. Default value is **true**, but you can set it to **false** if you prefer. Suppose the Oracle currently has N members (e.g. 1, 2…) and an event changes N to N'. If N' > N, this is always permitted. Thus if you had 1 Oracle member and add 1 to now have 2, no issue arises. ISIS_OracleSIZE and ISIS_HOSTS determine which processes are candidates to join the Oracle and how many the system will try to maintain (3 on any machines, by default). If N' < N, the behavior depends upon ISIS_IGNOREPARTITIONS. By default (**true**), such changes are ***always allowed***: if the Oracle had N=3 members but 2 failed, the remaining 1 continues execution. This is fine if the failures are real, but problematic if the failure was a network partition. In that case, we might now have a split brain situation, with two Oracle processes, each handling a subset of the system, and neither side talking to the other. With ISIS_IGNOREPARTITIONS=**false**, the behavior changes. Now the rule is that if N' ≤ N/2, the Oracle (more accurately, the side with the minority of members) shuts down and reports that it "lost the majority". The other side ideally has N' ≥ N/2 and remains live. The idea is thus that if we indeed had a split, the side where the Oracle retained a majority can advance; the other side shuts down. What happens later if the network partitions, but because ISIS_IGNOREPARTITIONS was true, the system continues operating on both sides? Eventually, when the partition heals, the two sides will be able to connect to one-another again. After about 5 seconds, the two sides will notice one-another, and as soon as this happens, one side will shut itself down, reporting that Isis detected a partitioning situation. The rule used at this step is somewhat arbitrary: we compare the addresses of the leaders of the respective Oracles, and whichever "wins" in this comparison gets to remain live. The other side shuts down. No attempt to merge state occurs.

	The only way to preserve state and merge it back in such cases is to have application logic that will keep the updates in a log, relaunch on the side that shut down, and replay those logged updates. (If you get lucky an entire group might have been active on the minority side, in which case if you used g.Persistent(), the group reloads its state when it restarts, assuming you've coded some kind of restart logic into your system). But if a group was split by the partition the challenge of merging state can be tricky. All of this makes it hard to use Isis in situations where partitions are a real risk. In a modern cloud computing data center, a partition is such a rare event that you probably don't need to worry about the issue at all… *internal to a single data center*. In a WAN, however, Isis would be challenging to configure properly because partitions really do occur in WAN settings, and Isis simply won't merge the two sides automatically. Our suggestion would be to run two instances of Isis separately, one on each subsystem in a WAN (e.g. one in each data center). Set the TTL so that they won't notice one-another. Then have some form of application-specific WAN logic to get the application groups to talk to one-another over WAN TCP links of your own. We would be receptive to ideas aimed at better solutions to this problem.
ISIS_IGNORESMALLPARTITIONS	ISIS_IGNOREPARTITIONS leaves one annoying case: if N=2 and then one member of the Oracle fails or just shuts down normally, we would shut the healthy member down too! This is because when we drop from N=2 to N=1, the healthy Oracle member thinks a partition might have occurred. We are doing two things about this. In a future release, we plan to handle shutdown failures differently from timeout failures. So this would already help: if N=2 and one member shuts down deliberately, it wouldn't trigger the partition sensing logic, no matter how you configure things. Meanwhile, if ISIS_IGNORESMALLPARTITIONS=true (and this is the default) then a transition from N=2 to N=1 is ignored even though this actually could be a partitioning event. The behavior is helpful when debugging, but you would probably not be wise to use this default setting in a mission-critical system where ORACLESIZE=3 and where N=3 Oracle members are normally running. In such a system a transition from N=2 to N=1 very likely would mean that a partition actually has occurred, and you would probably want the single isolated Oracle member to shut that chunk of Isis down.
ISIS_MAXMSGLEN	Tells Isis how large its UDP chunks can be, in bytes. By default this value is set to 64KB (e.g. 64*1024), but you can use larger or smaller values if you wish. Very old computer networks would have used 8KB, and there may still be some systems on which larger values trigger poor performance. On the other hand modern HPC systems sometimes permit huge UDP packet sizes of 1MB or more,

	and if you tell Isis[2] to do so, it will generate these large packets without chunking them.
ISIS_UDPCHKSUM	Tells Isis to enable the UDP checksum feature. Not needed unless the ISIS_SIGS feature is disabled.
ISIS_MCMDREPORTRATE	Tells Isis how often to recomputed the Dr. Multicast mapping
ISIS_TTL	Specifies the TTL that Isis will use in IPMC messages. This can be changed if necessary (the default is 0) but ONLY AFTER CONSULTING WITH THE NETWORK ADMINISTRATOR RESPONSIBLE FOR YOUR DATACENTER NETWORK. Inappropriate use of large TTL values can disrupt applications not using Isis[2].
ISIS_MAXASYNCMTOTAL	Limits pending unstable messages in the system
ISIS_DONT_COMPRESS	Tells Isis not to try and compress large messages
ISIS_TOKEN_DELAY	Delay in ms between sending tokens

Appendix1: Using Isis² to build a cloud service on Azure

As illustrated in Figure A.1 (a), a web service is usually composed of a Web front-end and a service back-end. While the front-end is centered on tasks relating to user interaction, the back-end handles hard-core application logic, including computation, I/O operation and other non-interactive functions. For example, in the now-famous iPhone "Siri" application, the front-end captures an utterance from the user, but then sends it to the back-end where a natural language speech application matches the user's statement, maps it to a set of requests that Siri is able to handle (for example, those relating to the calendar, web search, or to booking restaurants) and carries out the desired task.

To give another example, many kinds of web applications basically search a product set (such as men's dress shirts in size L), displaying available products. Here the front-end captures the search request but merely ships it to the cloud-hosted back-end. The back-end retrieves inventory information from the associated database. When the database operation is done, the back-end needs to send back results to the Web front-end, which then responses to the user. This pattern, then, sometimes in a much fancier form (servers often send relatively smart web pages, not necessarily just a picture of a shirt), dominates the modern web.

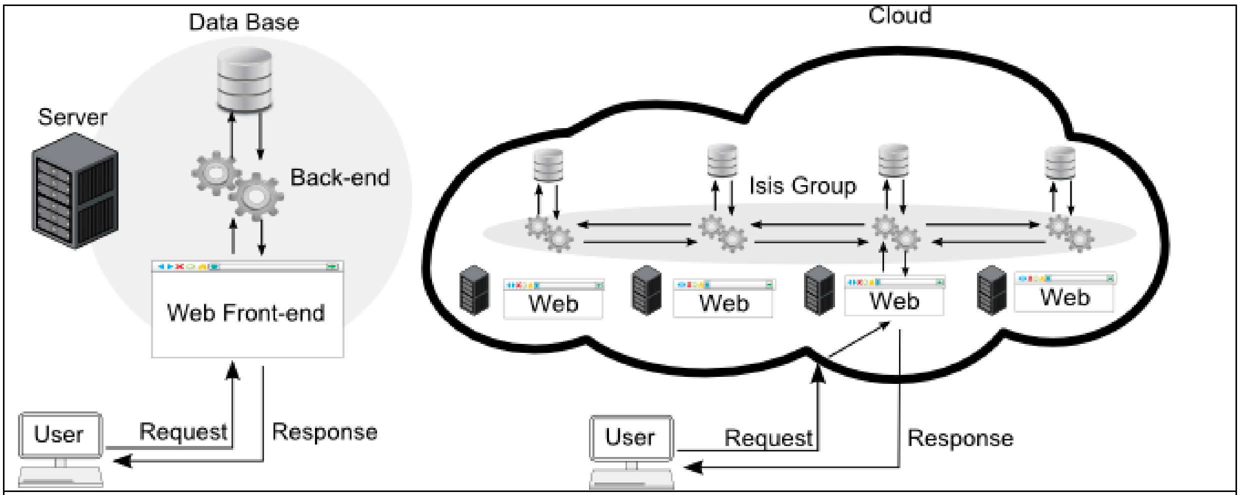


Figure A.1: (a) Inventory application in the form of traditional Web service; (b) Inventory application in the form of Cloud Web service, supported by Isis

In a classical computing setting, we tended to have one server for each client. In a cloud computing environment, the server side of this model still pairs each client with a single server, but we often have far too many clients for any single server to do all the work, and some tasks involve searching far too much data for any single server to do the search quickly enough. Instead, a potentially large pool of virtual hosts residing in a data center will provide computation and I/O operation, offering the client access to a big and resource-rich platform. If multiple requests are performed concurrently, the cloud platform will typically load-balance clients over the pool of resources so that each client gets rapid service.

We end up with the model seen in Figure 1 (b), where each cloud computing host is running a replica of Web front-end and back-end, perhaps with its own copy of the database (or some part of the database). When a user's request arrives, the cloud data center directs the request to a suitable host by using a load balance technique, perhaps combined with rules for mapping particular clients (or particular queries) to hosts that have responsibility for a particular subset of clients or data. Once the request reaches the target host, its front-end can work exactly as would a front-end built in a Web service model with a single server.

Thus the main opportunity to use Isis^2 is in the back-end service, where we might want to coordinate data replication or actions over a set of hosts to obtain parallelism or other functionality. The code shown below is a snippet of the required logic. By having each of the service instances run this logic, we are able to create one or more Isis groups that span the set of virtual hosts running on the cloud, and the group members can access the full set of Isis^2 operations.

Our first task is configure Isis^2 so that the members can find one-another. Here's how we would do that:

```
        static void Main(string[] args)
        {
             // Set up runtime environments
             Environment.SetEnvironmentVariable("ISIS_UNICAST_ONLY", "true");
             Environment.SetEnvironmentVariable("ISIS_HOSTS",
               "master.domain.com");

             IsisSystem.Start();
             SmallGroup smallGroup = new SmallGroup("Azure Group");
             smallGroup.Join();
             IsisSystem.WaitForever();
         }
```

In this particular example, we use environment variables to disable IP multicast, because the Microsoft Azure framework limits our services to talk to one-another using UDP. Notice that the code does this by manipulating the environment variables rather than by "reaching into Isis^2" and just setting the variables directly. In fact we normally wouldn't write code to access the environment at all; more typically, one uses the "SetEnv" command in the shell to set them prior to launching the service application. Any of these three options would work; the last is more natural and more standard.

```
        static List<int> rankList = new List<int>();

        static void OnReceive(IAsyncResult ar)
        {
             Byte[] receiveBytes = receiver.EndReceive(ar, ref web_);
             string receiveString = Encoding.ASCII.GetString(receiveBytes);

             switch(receiveString.ToLower())
             {
                 // Other cases are hidden due to the space limit
                 case "query":
                     {
                         rankList.Clear();
                         int nr = smallGroup.Query(Group.ALL, SmallGroup.myTO,
                            SmallGroup.COUNT,   1,   "Query",   SmallGroup.myEOL,
                            rankList);
                         string result = nr.ToString() + " returns: ";
                         int total = 0;
                         foreach (int count in rankList)
                         {
                             result += "| " + count;
                             total += count;
                         }
                         result += "| Total processors: " + total;
                         Console.WriteLine(result);
                         Byte[] tosend = Encoding.ASCII.GetBytes(result);
                         sender.BeginSend(tosend, tosend.Length, web,
                            new AsyncCallback(OnSend), ss);
                         sending = true;
                     }
                     break;
                 default:
                     {
                          // Do nothing
                     }
              }
        }
```

This next figure illustrates how a small group might receive and process requests that arrive through a web service API. Here we see a back-end service instance that distributes request (in this example, an inventory query) to all other processes in some group that all members join. When receiving the query, each process starts a concurrent (parallel) search within its local database records. The set of replies are passed back to the query initiator, and it in turn replies on behalf of the group to the external user

As seen in the code snipped, the logic is completely standard and uses the very same mechanisms that the manual discussed earlier. We see that the application first registers the Query request code (COUNT) and associated Query Handler. The code snippet above shows how the back-end service uses Group.Query API to distribute the Query, and gathers replies before sending it back to its local Web front-end. The code snippet below shows how to register a Query type and Query Handler function.

```
        public SmallGroup(string name) : base(name)
         {
             this.name = name;
             //Register a handler to a message (query) type
             this.Handlers[COUNT] += (myDels)delegate(int id, string keywords)
             {
                 try
                 {
                     // Search local inventory;
                     this.Reply(LocalInventoryCount);
                 }
                 catch (Exception e)
                 {
                     throw new Exception("COUNT exception " + e.Message);
                 }
             };
         }
```

As one sees from these small code fragments, building cloud web applications that use Isis^2 is really quite simple, and involves minimal effort beyond that needed to create a traditional Web application using the same platform (Azure, in this case). This particular example can be compiled and launched on Azure, at which point client applications running anywhere in the world can talk to the service via a web page or through a Web Services remote method invocation. They can send in a query (our example passes in an id and a keyword string, although in the example the id is always 1 and the string is always “Query”), the members do whatever they like to search the inventory, and then send back the LocalInventoryCount. The caller forms a result that lists the received counts, passing them back as a string.

A small remark about the types used in the Reply(): Isis^2 checks for type mismatches in most situations, but the quality of checking is more limited in the case of g.Reply(). Clearly, one would wish for g.Reply() to have a set of arguments that match the various reply-lists used by the Query caller. But in fact Isis^2 lacks this check right now. If the caller uses different reply types (and keep in mind that Isis^2 focuses on the true object types, not interface types or inherited types that can be used as synonyms…), a type mismatch is sensed and the reply isn’t delivered to the caller. For user-defined objects, you also need to use the same internal type ID number in the caller and in the group members (e.g. if the group members expect you to send in a PurchaseOrder, and the group uses TID=6 for this type, the client must also declare a class with the same marshaled data and registered using TID=6, just as the group members did).

Appendix 2: Installation Notes that you Really Need to Read

The present version of Isis^2 has been tested on Linux and Windows systems and can be used from C#, C++ and IronPython. It even works on the Android platform, under the Mono for Android library. The distribution is open source: as the user, you would normally download the source, compile with the appropriate compiler, and then use the resulting .dll file as a library from applications you code and compile using whatever development environment you favor (we use Visual Studio, but Eclipse would be another option, and there are really many choices). Installation instructions can be found at isis2.codeplex.com. We're in the process of revising this document to include code examples in each of the three languages just mentioned.

For example, a Visual C# user would either create a new "library" project, import the Isis.cs file into it and compile it (creating a .dll file), or create a new application project and add the Isis.cs file as a component of the application. If you built the system as a library, you would add a reference to that library in any application that later will use the library. You may also need to set the search path so that the debugger can find the library source code. One can then import the Isis2 namespace by means of a C# "Using" statement, at which point all the Isis2 API functions will work as described below.

The user must be careful to select an appropriate range of Internet port numbers for use by the system: these should be picked to be unlikely to conflict with other applications on your network. If more than one person is using Isis^2 in the same network, you'll need to take explicit steps to avoid clashes while debugging. Basically, two different users will want to use different, non-overlapping, ranges of port numbers, so that their applications won't end up trying to talk to one-another (which is how Isis^2 normally works, when applications notice other Isis^2 applications running).

The defaults are basically large random numbers and as far as we know, won't conflict with standard things you might already be running. You can change these values, either using the "setenv" shell command (this is the more standard way), or by just having your program modify them directly. The main control variables are ISIS_ ISIS_MCRANGE_LOW and ISIS_MCRANGE_HIGH (these numbers are defined with respect to the 24-bit "Class D" IPMC address range). As for port numbers, the relevant variables are ISIS_PORTNO and ISIS_PORTNOp: the first of these is used only in IP multicast mode (all Isis^2 IPMC groups use a single port number), while the second is the base value for a group of three ports that are used both in the normal UDP mode and for out of band transfers that run over TCP (the lower-case "p" is because ISIS_PORTNOp will be used for UDP traffic in Isis^2, called P2P traffic in some of our APIs, and inside the library there is also an ISIS_PORTNOa used for acknowledgements and an ISIS_PORTNOt used for OOB via TCP, as explained in the out of band fikle copying section of the manual, which you definitely want to read about because it offers the very highest speeds Isis^2 is capable of). Your firewall must not block these ports (on Linux your firewall is controlled by the iptables command, and you must use sudo to change it, e.g. "sudo iptables –F" to disable firewalls completely. On Windows, use the Windows Firewall command to configure this feature).

As explained further below, when Isis^2 first starts up, it needs to launch an “Oracle” membership service that would normally run on the first few machines on which the system is booted. The service runs as a separate subsystem within your application (on threads of its own that are spawned by Isis^2). The main role of the Oracle is to track membership of process groups and to assist in bootstrap for non-Oracle members. If the Oracle were ever to entirely fail (with no members left), all the non-Oracle processes will throw exceptions within 30 to 45 seconds. To avoid such issues, the Oracle can be replicated onto multiple machines. You can control how replicated this service will be, and can restrict the machines on which the Oracle can run. To do this, modify ISIS_OracleSIZE to the number of replicas you want (normally either 1, which won’t be fault-tolerant), and ISIS_HOSTS to the names of the machines you will develop on. By default, Isis^2 will hunt for other instances anywhere on the LAN.

If you try to start Isis^2 applications when the Oracle isn’t up, on machines not listed in ISIS_HOSTS, your applications will throw an exception. If ISIS_HOSTS is empty, the Oracle can run on any machines on which you run an Isis^2 applications. In this case Isis^2 automatically and silently decides if new Oracle members are needed and, if so, automatically starts the service when it has opportunities to add members. Since the Oracle runs “within” your applications, these opportunities arise when new application processes are launched (by you) and are eligible to host Oracle members.

By default, Isis^2 is configured to use IP multicast: it does so to find the Oracle, and also uses IPMC within groups for data replication, when updates occur. However, some systems disallow multicast and on those, Isis^2 can be configured to simulate it rather than using the hardware IPMC feature. You disallow IP multicast by setting ISIS_UNICAST_ONLY to true. With UNICAST_ONLY mode, the system sends packets purely using UDP. Isis^2 handles all of this automatically for you, so once you tell it what behavior you want, it should be transparent to you (except for performance impact of the settings you pick, of course). In ISIS_UNICAST_ONLY mode, ISIS_HOSTS must have a list of machine names on which the Oracle will be running. If you try and enable ISIS_UNICAST_ONLY but leave ISIS_HOSTS empty, the system will throw an exception: with no way to do multicasts, it would be unable to “find” other applications that are using Isis^2.

Things get *much* more complex if you want to use Infiniband or fast Ethernet over RDMA Verbs. For those you should look at the specialized parameters ISIS_NETWORK_INTERFACES, ISIS_USERDMA, ISIS_UNICAST_ONLY, ISIS_INFINIBAND and ISIS_FASTETHERNET. Email ken@cs.cornell.edu with questions. You’ll probably need a little help getting started in this particular case!

Optimizing Performance

Users focused on achieving the highest possible performance will generally try to run Isis^2 on a cluster (perhaps a virtual one created using an enterprise VLAN communications overlay), and will want to ensure that IPMC is enabled. But there are further tuning steps that can make a real difference to system performance.

One performance issue is that when sending normal messages, Isis^2 does a lot of work. C# is a strongly typed language, hence to generate a byte stream the system must cast C# objects into byte array representations, combine these into a single long byte array, perhaps sign it digitally, perhaps encrypt it, send it, and then repeat the sequence on receipt. So that makes normal Isis^{2-} message passing slow no matter how you try to tune it. We should be realistic about this.

But in contrast, the out of band file transfer can be very, very, fast. In this case we start with data mapped into memory in a file – hence the data is already in a byte array. (You need to put it there, of course, but a mapped file is just like any other file, plus you can treat it as a byte array if you like, so the costs can be quite low. Further, you can generate that mapped file in a program written in C++ or some other fast language, and Isis^2 won't even touch the bytes, hence even an application working with large numbers of massive files won't be particularly costly, provided that they fit in memory). Then you can configure Isis^{2-} to use some fast method to move the bytes. On Amazon EC2, for example, TCP is faster than UDP. Thus we have a parameter ISIS_OOBVIATCP (**true** by default) telling the system to move OOB objects via TCP when running with ISIS_UNICASTONLY=true. Otherwise the default would be to use IP multicast, which is even faster.

But the fastest of all would be to use one of the remote DMA transfer options: RDMA. RDMA is a new feature of Linux and Windows, and to use it you need to install O/S components that aren't present by default. So you need some expertise and will probably waste half a day setting things up. You also need a device that actually can support the so-called RDMA verbs mechanism, which the O/S level way of talking to an RDMA device. Examples would be Infiniband, and also Fast Ethernet (10Gb or 100Gb optical Ethernet), but only if your network card has an option called ROCE.

With these kinds of hardware, you can configure Isis^2 to do out of band transfers using RDMA over the fast network – direct memory-to-memory copying from the originating machine to the target. For example, imagine a C++ program that just ran inside Hadoop and created a new index of high value – perhaps the locations of all the cars in San Jose accurate to 5cm. You need to get this file to 75 machines that are running your spiffy new self-driving car service. Well, one way to do that in Isis^2 is to configure the system to use these fast RDMA network options (you'll need to own RDMA hardware and EC2 won't allow this…) but then your C++ program can talk to our WCF C++ server, asking it to "rereplicate" the file on the 50 target nodes. RDMA will be used to make the 50 copies, and the actual data will move purely through the hardware, memory to memory. Since the Isis^2 roles come down to sending some small multicasts with things like the file names, but not touching the bytes (we don't need to sign or encrypt the data in this case because Infiniband is a direct RDMA solution and nobody can spy upon it, and further is reliable, so no loss can occur), your gigabytes of data get moved at a speed of 20Gb/s or better.If this doesn't intrigue you, you are reading the wrong user manual!

But you very likely will need help from Cornell to pull this off, at least if you are reading this in 2014 or 2015. The technology is just very complicated to set up properly in the first place. It isn't our fault, really: RDMA seems to have been designed to be hard to use. Email ken@cs.cornell.edu and put "Help! I'm foolish enough to want to play with RDMA" in the subject. We'll help.

Down on earth, using Isis^{2-} in a more normal mode, our experience suggests that a dominating factor that can really shape the performance in major ways is the behavior of the underlying UDP communication sockets, when UDP or IPMC is employed. UDP performance, in turn, centers on the loss rate that occurs within the kernel. And this, in turn, is largely determined by the size of the kernel limits imposed on socket buffers.

To understand how these factors interplay, it is helpful to appreciate several related factors:

- Isis^2 itself imposes a maximum message size. No UDP or IPMC packet will exceed the value of ISIS_MAXMSGLEN, which is 32KB by default. You can modify this constant either by editing the code and recompiling (not recommended because each patch release would need to be changed again), or by calling Environment.SetEnv("ISIS_MAXMSGLEN=xxx") in your application, or by using the SetEnv command-line feature of the Windows or Linux shell. For example, you could set ISIS_MAXMSGLEN=4028 to limit the system to generating packets of at most 4KB. Conversely, on a system configured to allow it, you could set ISIS_MAXMSGLEN=16MB. You have considerable freedom here.
- Given a value for ISIS_MAXMSGLEN, the system will set the socket I/O buffer sizes. Isis^2 is harded coded to run with socket.RecieveBufferSize set to 64*ISIS_MAXMSGLEN and socket.SendBufferSize set to 32*ISIS_MAXMSGLEN, but read the remainder of this section, because it isn't quite as simple as you might think. Thus by default, Isis^2 is using a 2MB receive buffer size, and a 1MB send size.
- Next we run into the issue that operating systems limit the maximum size of the socket sending and receive buffers in the kernel, and loss will occur if these are exceeded. These limits are controlled by various boot-time and command-line options that can be modified by the developer, but that typically require root authority. If this limit is smaller than what Isis^2 attempts to use, Isis^2 will "think" the socket is configured with a larger buffer but in fact the cap imposed by the kernel will apply. This can have very strange consequences, and we'll explain how to check for them below.
- Finally, there is the question of how your underlying network behaves. If you use large UDP packets but the underlying network is limited to smaller IP packets (the relevant constant is called the maximum transmission unit or MTU and is limited both by hardware and by the O/S). Modern clusters generally support massive MTUs but that feature would often need to be expressly enabled by the cluster administrator. The general Internet limit is 1400 bytes (yes, a tiny number!) and by default, this is what you should expect. While the O/S will automatically break a larger UDP packet into segments that match the MTU size, if there is any underlying loss rate in the network or in the kernel, the more IP fragments, the more risk of loss. Typically, the rule is that the kernel must receive and reconstruct the full UDP packet (fragment by fragment) before it even considers the size limits on the socket that will accept that packet, and there is a stringent time limit imposed for this purpose; if it is exceeded, the entire UDP packet is dropped. Keep in mind that IP is an unreliable protocol, so if IP packets are dropped for any reason, the entire UDP packet will be lost.

Clearly, you'll see the least packet loss if Isis^2, the O/S kernel, and the MTU are all configured in a way that matches. But because the MTU is a property of the network itself, you can't really control that unless you are in an unusually good dialog with your network administrators. In contrast, you actually can modify the O/S limits and, as you've seen, you can configure Isis^2 to match those limits.

If you need to adjust the kernel limits on your platform, you have several options. The one that solves the problem once and for all requires that you modify /etc.sysctl.conf and reboot your Linux system. But you can also experiment with one-time changes:

On Linux execute (as root):

- sysctl -w net.core.rmem_max=2096304
- sysctl -w net.core.wmem_max=2096304

On Solaris execute (as root):

- ndd -set /dev/udp udp_max_buf 2096304

On AIX execute (as root):

- no -o rfc1323=1
- no -o sb_max=4194304
- no -o udp_recvspace=4194304
- no -o udp_sendspace=65536

Note that AIX only supports specifying receive buffer sizes of 1MB, 4MB, and 8MB, hence if you do use 4MB, you might as well also set ISIS_MAXMSGLEN=128KB. On the other hand, there is an issue with IBM's 1.4.2, and 1.5 JVMs which may prevent them from allocating socket buffers larger then 64K. This issue has been addressed in IBM's 1.4.2 SR7 SDK and 1.5 SR3 SDK.

Note: Windows does not impose a buffer size restriction by default.

One last remark: Isis^2 can be configured to use Infiniband as its basic transport. This does speed it up a little but much less than you might expect. The big win with Infiniband is for OOB transfers (see above, especially the remark about asking Ken for help). The same goes for Fast Ethernet: For the system itself, the default settings aren't bad. The big win would in situations where your data is in memory mapped files and you use the OOB code (out of band file transfer) to move bytes directly via RDMA from originator to destination(s) over the RDMA feature of Fast Ethernet, which is called ROCE. But we do support ROCE. And with our help, you can make this work.

Diagnosing Performance Issues

Before fighting with a cluster or network administrator who may be reluctant to grant you root access, it can be useful to see whether or not Isis^2 is actually suffering from any kind of performance issues on your configuration. For this, we recommend that you run an experiment and then print the value returned by Isis.GetIOState(). (This is one of several status dump options in the system; others include Isis.GetState() which prints a summary of the full system status on a process using Isis^2, and myGroup.GetState() which prints the state of individual groups).

The output of GetIOState() will look something like this:

```
Summary of network statistics:
    SENT: 17272 UDP (18435824 non-duplicated bytes), 0 tokens, 437 IPMC
(123668 bytes; 422 were stability packets), 16846 Acks, 65 Nacks.

    RECV: 14877 UDP (1877 were dups; 15890912 bytes), 0 tokens, 4633 IPMC
(1316804 bytes; 2646 were stability packets); 1877 were dups, 19418 Acks,
126 Nacks, 0 token-triggered resends
```

Shown here are the number of UDP packets that were sent by this member (17272) and the number of bytes that were sent, not counting retransmissions (18MB for this example), the number of "tokens" circulated if the g.SetLarge() feature is in use, the number of IPMC packets (these are really UDP sends but they will be received by multiple group members, so we show them separately: such a send triggers N receives if the group has N members), the number of these that were overhead packets, and the number of UDP packets that were actually acknowledgement or negative acknowledgements for incoming traffic.

Next we see the number of UDP packets received, how many were duplicates, etc. You'll see that the acknowledgement rates can be quite high; this is necessary for proper flow control in Isis^2 and is normal.

If you take a set of members and print all of their GetIOState() statistics, you can actually calculate the loss rates Isis^2 is experiencing. Overall, the total number of UDP sends and IPMC sends (multiplied by the group size) would be expected to equal the total number of receives. If a system is lossy you may see that two or three sends are being done for each successful receive.

In very rough terms, loss is the most serious cause of poor performance. After this comes inefficiency: sending many packets when one larger one could have done the trick. And after this are issues of scheduling and other overheads. Thus if you have a lossy system, recompiling Isis^2 with heavy optimization will have no impact at all, whereas if you've completely eliminated loss, optimization can pay off.

We're actually very curious to learn what the best way to handle really big data would be. If you happen to be in a position to use a large number of nodes in a more or less dedicated way, and can play with extremely large MTU and kernel limit values, drop us a note and we will be very happy to help you. We're wondering what the highest performance rates a system like Isis^2 can achieve might be, and what mixture of settings would achieve those bounds. Indeed, we would love to write a paper on "Data Center Gigacast with Isis^2", looking at the best way to configure the system to support groups within which replication of gigabyte-scale objects is required.

Getting back to earth, here are some other mundane options to consider. By using the "netstat –a" command and the "pstat" command, or the Windows performance monitor, you can also monitor kernel level counters maintained within the Linux or Windows system that track UDP packet loss. This gives a second way to catch events that might be triggering data loss. Drop us a note if you see something very strange and clearly tied to your use of Isis^2.

Finally, a bit of pragmatic advices. Very commonly, rather than actually work all this math out, one simply tunes the various parameters and uses GetIOState() to quickly check to see if things are improving. By running the identical application to completion and then comparing the GetIOState() outcome data, you can quickly see whether your tuning is helping, hurting, or having no impact. You can also compare a Windows and a Linux run of your program, looking for noticeable problems. This is especially helpful if you have the same program running on the same hardware in both Windows and Linux and are seeing different performance. Small differences can occur because, after all, these operating systems aren’t identical. But if you saw a 3x slowdown on one platform compared to the other, you should immediately guess that something is causing a high loss rate and a high retransmission rate.

Appendix 3: How Does Isis^2 Overcome the CAP Theorem?

Isis^2 is optimized for high-speed applications deployed on a large scale, typically in an enterprise LAN or a data center (we can support WAN deployments too, but not if partitioning failures are common). You may have read about the CAP principle (it claims you can have just two of "consistency", "availability" (rapid responsiveness) and "partition (or fault) tolerance". CAP comes with a theorem, and many data center developers have accepted the underlying argument that the only way to get speed on the cloud is to abandon security and consistency guarantees.

Our experience with Isis^2 suggests otherwise: CAP applies on platforms that don't fully leverage multicast, but when multicast can be used to replicate data at very high speed, not only does one discover new paths to very strong consistency, but the resulting highly assured applications can actually execute at higher performance levels than in more standard cloud platforms, where data replication is typically via some form of point-to-point chaining and hence slow. This could be true IP multicast (which can be used safely in Isis^2) or it could be our built in *emulated multicast* which runs over unicast (UDP) packets.

The properties Isis^2 guarantees are, somewhat paradoxically, the key to why CAP doesn't apply to it in any simple way. CAP revolves around a problem with replicating data: in modern data centers, nobody trusts IP multicast, and without IP multicast, replication is slower than molasses. Worse still, because of the way many systems handle failures, in traditional systems the only way to ensure that a service replica is reading the correct value of data is to read multiple copies and compare their revision histories. This problem is quite serious in data centers where one replicates tasks to handle huge numbers of client queries: after all, replicating a task would seem to entail replicating the data on which that task operates, which in turn implies replicating any updates. If updates propagate slowly, applications will often be operating on stale data, hence may suffer a loss of consistency, which is just what CAP argues.

But if updates propagate rapidly, as they do with Isis^2 when it runs over IP multicast, this issue isn't seen and the option of offering high assurance at high speeds becomes practical. In Isis^2 we never need to pause to do a "quorum read" or a "quorum write", and have written a paper on this point:

Overcoming the "D" in CAP: Using Isis2 To Build Locally Responsive Cloud Services. Ken Birman, Qi Huang, Dan Freedman. Submitted for publication, May 1, 2011.

The basic idea is this: As in any scalable system, we recommend that you think in terms of *hard state* of the kind stored in databases and files and *soft state* such as cached data that can be regenerated if necessary. We then offer one execution model (virtual synchrony) but two suites of protocols that support it. One is for hard state; these protocols are much like Lamport's widely known Paxos technology. The second suite is for use with soft state and scales and performs extremely well. Yet the same model applies: the difference is a bit like what an optimizing compiler does when it realizes that data is in a register and, knowing this, generates simpler, faster code.

It will turn out that the performance difference between these primitives is substantial: Send, used to update soft-state, gives nearly flawless scalability in our tests on up to about 1000 group members, but SafeSend, used for hard state, slows down linearly with the number of group members and also starts to show very high packet loss rates and erratic latencies. Thus it will be important to use Send in large-scale groups, and to avoid hard-state in those groups. This doesn't preclude having separate hard-state services that are consulted from time to time, but the most scalable, highest performance parts of your application should avoid hard-state. Think of your application as having a soft-state front-end that shields the hard-state back-end components by reducing the load the experience; this approach will pay off with far better scalability and speed than you might imagine possible. In contrast, if you put hard-state up front, your service will be sluggish even at modest scale.

The rules are fairly simple, but we do recommend reading the more detailed paper to understand them in depth.

- State is *soft* if a soft-state replica that crashes would recover it by means of a state transfer from a live replica (we're not talking about cases where a whole group shuts down and later restarts; we have in mind situations where *f* out of *N* members crash and *N-f* continue.
- Hard-state would normally have an associated file or database representation, with each replica owning a replica of the file or database, and a restarting replica would want to resume using its local copy of the file or database.

In a nutshell, applications can use Send or OrderedSend to update soft state. Send guarantees FIFO ordering (messages are delivered in the order they were send, just like with TCP). OrderedSend is a bit more expensive and guarantees that everyone receives every message in the same fixed order, even if concurrent multicasts are sent by different senders.

Be aware that there is a brief window during which a Send could be "lost": delivered at some system members but then erased by a subsequent crash. Accordingly, prior to sending responses to an outside user, it is best to call Flush if there were any prior Send or OrderedSend updates on which that response depended. The window of vunerability for Send and OrderedSend is very short (it normally would not exceed a few hundred milliseconds). Once a Flush completes, any prior Sends have been delivered, which is why calling Flush prior to talking to an outside user provides useful protection: the user would never perceive the system as making some statement, but then developing amnesia by forgetting some part of the system state on which that statement depends.

In contrast, use SafeSend to update hard state. SafeSend is just like the famous Paxos protocol by Leslie Lamport: it delivers messages in the same order everywhere, and also durable, in the sense that the messages can't be delivered to some destinations but then erased by a crash. SafeSend is definitely very safe, but can be quite slow. However, we should perhaps point out that SafeSend is slightly tricky in one sense: because the notion of durability is application-specific, you'll need to configure two parameters; one tells the system how many copies are needed to achieve the durability threshold, and the other is a handler, called via upcall, that logs these copies in whatever manner you wish to achieve durability. The default does in-memory logging and hence is fast but not all that secure. A pre-built module that integrates with the Isis^2 group logging feature is available; it logs to disk, and replays any interrupted SafeSend multicasts after recovery from a total failure, with some small risk of delivering an update twice (you'll need to check for this and filter out the duplicates if this would cause problems for your application). And of course you can also add a new application-specific durability module if you prefer.

With all of its multicast primitives, Isis^2 guarantees the virtual synchrony model. We think of Send (and OrderedSend) as optimized versions of SafeSend: soft state doesn't need durability, so Send doesn't guarantee it. If you do a call to Flush before replying to external users, the reply to your external client won't be sent unless any prior Send operations have become durable: a quick way to obtain something very much like SafeSend.

This sequence would be smilar, but not identical to SafeSend. To really get an identical behavior you would need to use Send to send a burst of multicasts, then would call Flush to make sure the Send operations are done, and then some sort of Commit to tell all group members that the Sends are finished (until receiving a commit, they should buffer but not process those prior updates). After sending the Commit (and if you wish, doing one more call to Flush, it would become safe to reply to the user. But if Paxos is what you want, don't go to the trouble of implementing anything so elaborate: call SafeSend and save yourself the hassle.

When (and why) is it Safe for Isis^2 to Use IP Multicast?

We mentioned in the introduction that Isis^2 is option you can disable this feature entirely (and would need to, on a platform like Microsoft's Azure or Amazon EC2: they don't permit IP multicast use by applications at present). But if you were to research the topic, you would quickly learn that data center operators are very uncomfortable with IP multicast. How might you convince your local data center operator to allow Isis^2 to use this dangerous technique, even if your enterprise won't allow other systems to do so?

Many data centers avoid IP multicast because of fears that it can behave erratically under heavy loads. We've studied this problem very carefully, working with IBM researchers who had extensive experience with challenging cloud computing scenarios in which IP multicast was known to break down. This work revealed an explanation: it turns out that modern routers have a limited capacity for IP multicast addresses. The issue is tied to the way that data centers route IPMC messages: to route at full speed, they use a kind of a hashing scheme called a Bloom Filter to decide which links need a copy of each IP multicast. And these filters have just a few thousand bits. But the IP multicast address space can have 2^{24} unique addresses in it, which is way more than any router will dedicate to a Bloom Filter bit map. The filters fill up and IP multicast becomes more like "data-center-wide broadcast", overloading everyone with unwanted messages and triggering massive loss. (There is a very similar issue in the network interface cards on the individual hosts, which also get overwhelmed, so the filtering won't even occur in the NIC: it has to be done in software by the operating system!)

With IBM we published a paper on a mathematically-rigorous remedy to this issue, *Dr. Multicast*:

Dr. Multicast: Rx for Data Center Communication Scalability.
Ymir Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, H. Li, G. Chockler, Y. Tock
Eurosys, April 2010 (Paris, France). ACM SIGOPS 2010, pp. 349-362.

This system allocates IP multicast addresses just to the largest most active groups, and also merges similar groups. These steps yield dramatic reductions in the numbers of IP multicast addresses needed to serve a given application. But then we go further, and impose a hard limit: Isis^2 will use no more than the number of IP multicast addresses the system is permitted to employ, and the default (if you don't override it) is 25. So IP multicast doesn't melt down, and we're halfway to our goal. Groups that are too small and too slow to merit a true IP multicast address use point to point UDP to send their messages; we do this automatically and the user doesn't see a thing.

You can control the use of IP multicast through various parameters (see the table at the end of this document), and also by calling the per-group APIs myGroup.UseUnicast() or myGroup.UseIPMC(). The former forces the group to use point-to-point UDP, while the latter encourages Isis to assign this group a true IPMC address.

The other part of the story is that when using IP multicast, if a process receives from too many senders, it can drop data just because the aggregated data rate can become very high. Isis^2 allows you to specify a per-group rate limit so that if you are aware of such a risk, you can prevent loss. By default we don't activate this rate limit, simply because the problem impacts just the application that misuses multicast and overloads itself. But the mechanism is there for you if you need it.

Note: some systems combine small multicast messages into larger ones to get a further speedup; the effect is similar to writing files a block at a time rather than a byte at a time, which has a *huge* impact in file I/O performance. At the network layer, where the underlying packets are typically limited to 1400 bytes, the value of consolidation of this kind is less dramatic than in a file system. Nonetheless, with very small user objects, sending multiple objects in a vector can be a big win (e.g. instead of sending 10 updates that each list a name and a new salary, consider grouping them into a vector of 10 names, and a vector of 10 salaries, or even a vector of 10 name/salary pairs implemented by a new class of your own). Isis^2 supports object vectors in messages, and encodes the data efficiently, especially for byte vectors of basic data types like int, double, etc.

But just for clarity, we should again emphasize that you can use Isis^2 even in settings that disable IP multicast entirely. The way to do this is to either set the ISIS_UNICAST_ONLY flag. Performance will suffer, but Isis^2 has a reasonably efficient internal overlay scheme to mimic multicast over direct UDP packet sends, and the model itself will still be available; you won't need to change your code at all.