Hosting Dynamic Data in the Cloud with Isis2 and the Ida DHT

Ken Birman and Heesung Sohn
Dept. of Computer Science, Cornell University

Abstract

The big-data community generally favors a two stage methodology whereby data is first collected, then uploaded for analysis using tools like MapReduce. During analysis the data won't change; this simplifies fault-tolerance and makes it worthwhile to cache intermediary results. In contrast, when it is necessary to capture data continuously and query it on the fly, cloud storage and access technologies must be reexamined. Isis2 aims at such scenarios, offering a base set of mechanisms that replicate data and perform computation with strong consistency and other assurance properties, then layering higher level abstractions over this core. Here we a focus on a subsystem called the Isis2 *interactive data analysis infrastructure*: Ida. Ida is a strongly-consistent distributed key-value store on which surprisingly complex computational tasks are feasible.

1 Introduction

Our effort aims at a limitation of existing cloud-computing infrastructures: aggressive lock-free data replication and caching has been a key enabler for dealing with huge numbers of clients, but at a price. While applications achieve extremely rapid response rates, they also exhibit frequent transient anomalies. The approach must be revisited if the cloud is to host a new generation of "mission critical" applications that perform data-parallel tasks in real-time and require stronger assurance properties. For example, it is likely that cloud-based systems will be needed to manage the future smart power grid. The safety and security of the power infrastructure will thus depend upon the integrity of the cloud-hosted applications used to monitor the grid and initiate appropriate actions. Future cloud-hosted medical platforms will monitor high-risk outpatients (such as at-home diabetes patients) and provide forms of automated care (such as dynamic control of insulin pumps). Self-driving vehicles will depend on cloud data to avoid hazards.

TRIOS'13, November 03 2013, Farmington, PA, USA

http://dx.doi.org/10.1145/2524211.2524212 ACM 978-1-4503-2388-8/13/11.

These examples center on a style of computing in which updates are continuously applied while data-parallel queries are concurrently issued against the data. Notice further that whereas MapReduce and similar tools often host long-running computations, these demanding "machine in the loop" scenarios would more likely be dominated by short queries and updates. Inconsistencies could pose serious risks.

The Isis2 system focuses on this style of computation. The system starts by offering scalable support for *state machine replication*: process groups and ordered, reliable group multicast used to support strongly consistent replicated data, concurrent computation, and coordinated fault sensing and reporting. A variety of higher level tools are layered over this core, including support for persistent external data sets (Paxos), migration and replication of large memory-mapped objects, locking services, etc.

We believe that the popularity of DHTs reflects their predictability, scalability and high speed. Here, we focus on the Isis2 DHT: a subsystem called Ida that offers stronger assurance properties without violating these basic DHT characteristics. Ida supports:

1. A generalized in-memory DHT model supporting insertion of multiple key,value tuples as a single action, and allowing user-specified handling of Put collisions (same key, different values).
2. Support for queries that span multiple shards. Results can either be sent directly to the initiator or aggregated using user-specified aggregation logic.
3. DHT members can access their local slice of the key-value data using code written in Microsoft's Language Integrated Query model (LINQ).
4. Excellent scalability and performance, with 1-hop routing: requests are sent directly to the DHT members that will process them. Concurrent events that touch disjoint DHT members won't interfere with one-another.
5. Fast self-repair. Ida uses node-rank in a *membership view* to enable 1-hop routing. The implementation replaces failed low-rank members with healthy high-rank ones, facilitating local repair.

Isis2, including the Ida DHT, is available for open-source download [1], and can be used from C#, C++ and Python on Windows or Linux platforms.

2 The Isis2 Platform

The Isis2 system updates the *virtually synchronous process group communication* [2] model for the cloud. The core construct is a group of objects residing within processes linked to the Isis2 library; new processes can join at runtime (loading current state from a checkpoint via state transfer), or leave (at will, or by failing). Ordered reliable multicast is used to transmit updates. When group membership changes a *new view* event occurs, ordered relative to other events. These new view events are synchronized relative to state transfer (the delivery of a checkpoint to a new member), to ensure that the new member starts with correct data.

Isis2 supports quorum-style protocols such as Paxos, but our emphasis is on a style of replication in which all updates reach every replica. This allows reads to be performed by any single replica. If multiple members replicate the same data, they can cooperate to concurrently perform costly tasks with guarantees of consistency. Thus, one can easily implement algorithms that build on a state-machine replicated abstraction, but also leverage the consensus view of dynamic membership to perform concurrent computations or coordinated actions.

The system offers an object-oriented API, using a polymorphic event-upcall model that will be familiar to users of other common cloud-computing tools. For example, much as a GUI might have an upcall for mouse-over, Isis2 has upcalls for new group views, multicast and unicast delivery, etc. The execution model, virtual synchrony, has been integrated with the stoppable state machine model [2], permitting Isis2 to offer developers the choice of a number of "flavors" of virtually synchronous protocols differing in their ordering and durability guarantees, including Paxos. IP multicast is used when feasible, and Isis2 manages the IP multicast address space to avoid overloading routers [29]. If multicast is not permitted, the system constructs an overlay mesh of TCP links, then emulates point-to-point messaging by routing within it.

Figure 1 illustrates the resulting system architecture. Isis2 has a lower layer consisting of probabilistically convergent components, many of which use gossip-style protocols (the bricks seen in the bottom layer on the left side of Figure 1). A higher layer implements reliable multicast.

An Oracle service manages membership and normally runs on 3-5 members; we launch it silently and it lives within the first few members that use Isis2. Among its roles, this service implements the *primary partition* model: if a network link fails, progress is permitted in just one partition. Processes in other partitions must wait until the link is repaired.

Many components employ complex protocols. It would be out of our scope to discuss those here, but the upshot is that whereas previous virtual synchrony implementations rarely ran "directly" on more than a hundred machines at a time, Isis2 runs on thousands.

The system is also quite fast. In [5] we undertook a side-by-side comparison of one of the core multicast primitives (virtually synchronous Send) with two configurations of Paxos (virtually synchronous SafeSend configured to use different numbers of "acceptors"). Our evaluation focused on reads and updates to an in-memory data set replicated on as many as 900 machines, and showed that if virtually synchronous Send is combined with an Isis2 primitive called Flush, it offers guarantees identical to Paxos for

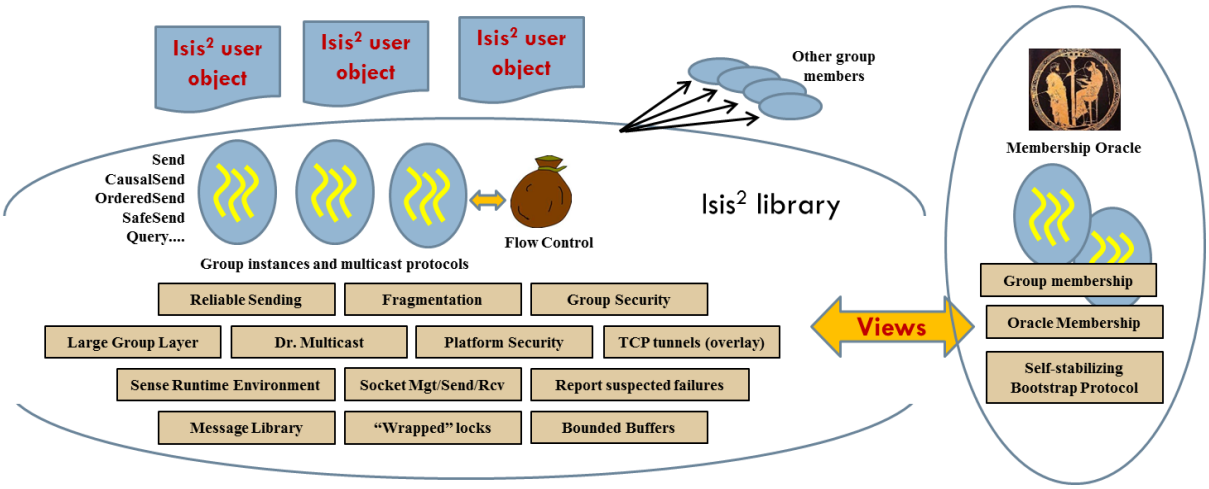

Figure 1: The Isis2 library implements virtually synchronous process groups and multicast for the cloud. Ida is one of a set of higher-level structures we've layered over it.

this in-memory case, yet achieves far better performance and scalability. This is just one of many situations in which the ability to pick exactly the right multicast guarantees yields dramatic benefits.

Layered over Isis2 is a set of higher-level packages that hide details, offering end-user functionality:

- A locking package (analogous to Chubby [6]).
- A tool for "out of band" migration and replication of large memory-mapped binary objects (like file shuffle and replication in Orchestra [9], but using IP multicast and offering stronger guarantees).
- DMake, a tool for monitoring and management of cloud applications that do not directly use Isis2.
- The Ida DHT, which is our primary topic here.

3 Strong consistency in a DHT

The introduction of strong consistency models into DHT-like platforms has been a topic of debate in the cloud community. The core dilemma centers on the required properties of a DHT: one selects a DHT rather than a database or a file replication tool to take advantage of the low overheads, high performance and scalability of the model. Thus, while our intended applications motivate us to extend the DHT model, Ida must retain several core properties:

- The operations in the Ida API all have "expected constant" cost. Here we should emphasize the word *constant:* some DHTs have a log(N) routing delay, but those used in cloud settings typically need 1-hop routing: a client sends requests directly to the DHT member(s) that will handle it, with no intermediary forwarding. This translates to low latency, a critical DHT property.
- Minimal disruption when membership changes. Cloud DHTs must not degrade due to "churn". Adding members should increase capacity without increasing overheads.
- Costs must be localized: if an operation impacts just some subset of DHT members, only that subset should have work to do.

One could use a cloud database system such as Spanner [10] as if it were a DHT, but such a solution wouldn't have the desired properties: Spanner offers a full transactional API, giving it great flexibility, but is far less predictable and optimized for high aggregate throughput. Individual operations may experience significant and unpredictable latencies. The developer who rejects a high-functionality database such as Spanner in favor of a DHT is probably doing so to be certain that operations like Get and Put will map to a single RPC, and would only be delayed if a hot spot forms. On the other hand, nothing about the DHT model demands inconsistency, and it is easy to identify use cases needing stronger properties:

- Key-value pairs could represent the state of an autonomous vehicle, a mobile user, or a physical infrastructure like the smart power grid. One could then run distributed machine-intelligence algorithms directly on the DHT to make decisions. Consistency enables decentralized execution of the associated logic, so that even with a rapidly changing knowledge base each decision will be based on a state that really existed at some instant in logical time. Lacking consistency, the application might observe phantom, non-existent states that would trigger unsafe actions.
- Many social networking and web algorithms compute on graphs or similar structures. One can represent these as key-value pairs; when mapped to a DHT, the costs of actions would then be predictable. Notice that a "value" could include the page weight or rank, the time when the page was last scraped, etc. It is important to access such a graph along a consistent cut; otherwise, problems such as "phantom cycles" arise [8].
- Key-value pairs could represent the state of a multiuser game, or a virtual reality environment; here consistency translates to agreement on the game state, agreement by a group of friends on where they will meet after work, etc.

A DHT model allows the developer to spread data widely in an elastic manner, and to leverage parallelism when computing on it. The predictably low costs translate to predictably low end-user response delays. Provided that this performance is maintained, strong consistency guarantees let us view the DHT as a representation of some "single" system state that evolves through time by a series of atomic transitions.

We are not the first to recognize that stronger DHT guarantees could be valuable. A number of prior DHT consistency proposals have used strong protocols on a per-shard basis, employing Paxos (guaranteeing total order and durability) [20][26], or even Byzantine Agreement (yielding resilience even to compromised nodes) [28]. But these only allowed updates to a single key-value item at a time. The use cases we have in mind would often represent some form of structure in the DHT, using a *set* of key-value tuples. Consistency for a single shard at a time won't guarantee consistency for structures that span multiple shards.

In Ida, the DHT APIs are generalized to allow a single request to atomically update or query multiple key-value pairs, giving rise to multi-shard operations that access a set of participants determined by the set of keys touched. Notice that because the set of keys will vary for each request, the participant sets will also vary, operation by operation. This is significant because the group communication protocols available in Isis2 prior to our work, as well as the protocols used

in the prior work on DHT consistency, were defined over stable or slowly changing participant sets. In Ida, the "group" that participates in each operation will be defined on an operation by operation basis.

Thus, we'll need a protocol that can carry out operations that overlap by triggering work at some of the same DHT nodes. But ordered multi-Get and multi-Put actions aren't sufficient. A further issue is that once we begin to talk about representing non-trivial structures in a key-value form, the simple replacement semantics of the standard DHT model no longer suffice. In a typical DHT, each new Put simply replaces any older key-value pair with the same key. With Ida, a new Put sometimes introduces a value that should be "combined" with the prior one in an application-specific manner (for example, added to a list, averaged in, etc.). This leads us to allow developer-specified methods to handle collisions, an idea introduced in the Piccolo DHT [25].

The Ida consistency model extends to this full range of scenarios, yielding a simple programming style in which the ability to assume consistency eliminates the need for the client application to worry about DHT errors that might be confusing or trigger inappropriate actions. Our design will sometimes require that the client issue a very small number of successive operations, but we assume that in all cases, that client is counting upon guaranteed rapid response.

Although an individual DHT member holds data for just the shard to which it belongs, the key-to-shard mapping will map many keys to each shard. Thus a single node could host a very large set of key-value tuples, and the question arises of how best to query the data. Ida supports multi-Get, but also allows a query to be initiated by *multicasting* a request to the participants, each of which then executes a LINQ (Language Integrated Queries) query on the DHT "slice" associated with its portion of the data. LINQ is an SQL-like technology available within .NET languages like C# or Python. The results can then be sent back to the query initiator or stored back into the DHT as new tuples.

Ida enables use of LINQ by exposing a slice of the DHT as a collection that has specified key and value types. Thus, the developer works with a model in which one can either fetch the data for some set of keys, or can multicast a request to the shards where those keys reside. The DHT representatives for those shards can then perform data-parallel LINQ queries, each on a slice of the DHT contents. If the initiator doesn't know the exact set of keys that are needed, the query could be sent to a covering set of shards, or (in the limit) the entire DHT. Then each recipient performs its local share of the computation.

4 Details of the Ida API

In this subsection we provide a more detailed exposition of the Ida multi-tuple API, its integration with LINQ, and the Ida aggregation infrastructure. Ida was implemented as an extension to Isis2. The basic DHT API is shown in Figure 2. To activate the DHT a developer first creates a group, then each of the members calls DHTEnable, specifying the intended typical group size and a desired shard size. For maximum stability, Ida DHTs often include spare members. For example, one could create a group of

void g.dhtEnable(int shard size, int target dht size, int ttl): Configures a DHT
void g.Put<KT,VT>(IENumerable<KT,VT> kvlist): Inserts a set of key-value pairs.
void g.OrderedPut<KT,VT>(IENumerable<KT,VT> kvlist): Same behavior as dht.Put but with strong consistency.
VT g.PutCollisionResolver(KT key, VT v0, VT1 v1): User-defined function that overrides the default collision behavior; it takes a key and a pair of values, and returns new value to be stored. The "new" value type need not match the "old" value type, and one can define multiple resolvers for different types.
IEnumerable<KT,VT> g.Get<KT,VT>(IENumerable <KT> keys): Returns a list of key-value pairs, one per key (if a key is not found, that item will be omitted from the list).
IEnumerable<KT,VT> g.OrderedGet<KT,VT>(IENumerable <KT> keys): Same behavior as dht.Get but with strong consistency.
IEnumberable<KT,VT> g.OrderedQuery(QueryKey QK, arguments to the query); DHT query.
void g.OrderedSend(QueryKey QK, arguments to the query): Initiates an aggregation query.
IEnumerable<KT,VT> g.Contents<KT,VT>(): Performed by a DHT member upon reception of an OrderedSend, this API offers access to the collection of key-value tuples available at that member, filtered to match the specified key and value types. One can use this list in Linq query expressions, etc. Having computed a contribution, the member calls dht.AggregationSetValue<KT,VT>(QueryKey QK, VT value).
VT Aggregate(QueryKey QK, VT v0, VT v1): User-defined aggregating function that takes a key and a pair of values, and returns new value that combines the two given values.
IEnumerable<KT,VT> g.AggregationWait(QueryKey QK): Waits for the result of an aggregation.

Figure 2: Ida API (KT designates some key type, VT a value type, and g a group hosting the DHT).

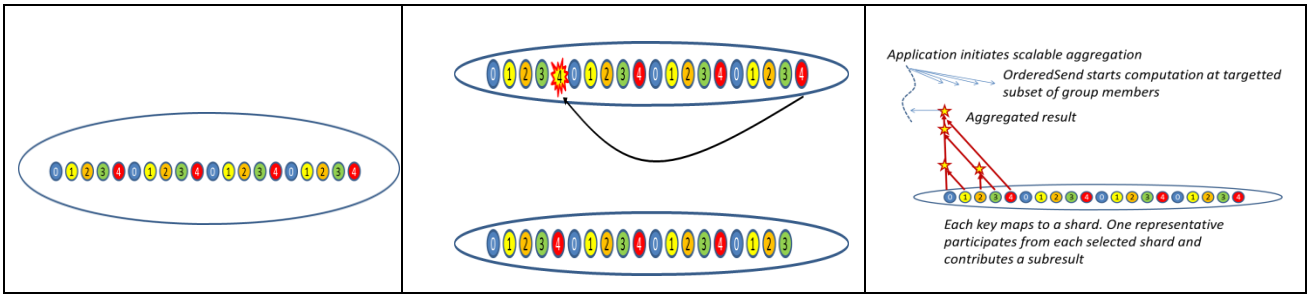

Figure 3: A group of 20 members in 5 shards (identifiable by shard-number and color). Each small circle represents a DHT member: a process running on some node within the data center. Ida targets groups that could have many thousands of members.

Figure 4: With 1-hop view-based DHTs, a failure can renumber the nodes, changing the DHT mapping. In Ida, an existing node replaces the failed one, thus limiting the impact of churn. If needed, state transfer brings the replacement up to date.

Figure 5: Aggregation Tree: After a query is sent via ordered subset multi-cast, one node participates for each shard to which a key maps. Here we see a query that touched all shards. The stars are invocations of an application-supplied aggregation method.

1005 members, but use target size 1000 and shard size 10. We would then have 100 shards, a few containing extra members (specifically, shards 0 through 4). Figure 4 shows a case with 5 shards.

Ida has built in handlers to implement the basic Get and Put operations, but issues upcalls to user-specified methods in more complex situations. For example, consider the delivery of queries to participants that will execute a LINQ-based operation. Here, the initiator employs an atomic multicast, OrderedSend, with a strongly-typed set of arguments. Ida uses an upcall to deliver the multicast to the appropriate members in the appropriate event ordering. The handler can then invoke mydht.DHT<KT,VT>() to access the DHT, after which it performs any desired LINQ operations on the collection of key-value tuples so obtained.

Two other situations in which user-specified plugins are employed involve the *put collision resolver* and *query aggregation* mechanisms. We say that two Put operations *collide* if they use same key for different values. In Ida, when a collision is sensed the default behavior is to replace the older value with the new one. However, if a collision resolution method is defined by the user, that method will instead be invoked. For example, one can maintain a list of values by inserting single items using List<ItemType> as the value type, and have the PutCollisionResolver merge new values into the existing list. In effect, the collision behaves like a message, and triggers application-specific logic.

While the idea is simple, it is perhaps not surprising that few existing DHTs offer it (Piccolo [25] does have such a mechanism). Collision resolvers are useful primarily when representing non-trivial data structures, but without a consistency model, it can be surprisingly hard to work with DHT representations of data structures that involve multiple key-value tuples. These kinds of structures would often seem "broken" during periods when updates are underway. Strong consistency is what makes a collision-resolution mechanism useful. Given the two features side-by-side, it becomes easy to represent a wide variety of data structures in key-value form (including elaborate structures, such as graphs).

The next situation in which user-specified code is called from the platform arise during queries that access large numbers of nodes. Within Isis2 the "normal" way of performing a query involves use of a method called OrderedQuery. OrderedQuery starts with an OrderedSend: a multicast to the recipients, which compute subresults and then send them directly to the initiator as point-to-point replies. In a large DHT, however, the many near-simultaneous incoming messages could overwhelm the initiator, triggering loss and other inefficiencies. Moreover, if combining partial results into a desired outcome involves work, a 1-to-all send followed by an all-to-1 response misses a chance to leverage parallelism.

Accordingly, Ida offers the option of switching to a different style of response in which an aggregation tree is employed to collect and combine the responses. For an aggregation query, the query is still sent to the shards using OrderedSend, but now it triggers a parallel upcalls to the query handler methods. Meanwhile, having sent the query, the initiator invokes AggregationWait with the QueryKey to wait for the result. As the query requests arrive, each participant computes its contribution to the request in parallel, then passes the partial result "up" the aggregation tree by calling AggregationSetValue(), again using the QueryKey as an identifier. Upcalls to the user-defined aggregation method are used to combine pairs of intermediary values. A wave of aggregation ensues, starting at the leaves and converging at the initiator: the root of the tree.

We see this in Figure 5: The first two partial results are combined by upcall to the aggregation method in the left-most node in the tree, while representatives in shards 2 and 3 compute a second partial result. Since

we have 5 shards, an odd number, shard 4 contributes a result of its own. The first two partial results are then combined, and finally the third partial result is combined with the intermediate result, giving the desired answer. Within the tree, each time a pair of partially aggregated values associated with the same QueryKey reaches an inner node, that node combines them by calling the user-provided aggregation method (the event is shown as a star), then passes the resulting key-value pair upward. This terminates at the root, where AggregationWait() returns the result.

Notice that in contrast to a database, where a query could potentially compare all the accessed tuples with one-another, an Ida query is done as a single parallel action, and the first steps are limited to the local slice of the DHT contents. The only opportunity to combine data from multiple shards arises when the local results are aggregated. Thus while the leader can distribute any shared information it wishes, an iterated computation would be needed if we wanted to use data from one shard as an input to the computation at some other shard. Ida certainly supports this, but the atomicity of its operations wouldn't cover a multi-step computation. We'll have more to say about the issues that arise in this style of computation in Section 7.

No matter how the results are collected, any Ida query can include action by the full membership of the relevant shards or even the full membership of the whole DHT, as in the example shown in Figure 5. On the other hand, not all requests need to involve the full shard membership. Recall that in Ida, data is replicated with strong consistency properties. Thus for read-only tasks, one contribution per shard would normally suffice. Moreover, only keys used in the query need to be examined, hence only shards corresponding to those keys need to be included.

Accordingly, the QueryKey is used not just as an identifier for the query, but also to track the keys and shards that will contribute to it. In this mode, Ida automatically selects one representative per shard, and delivers the request only at selected members. Notice that any single key-value pair contributes exactly once to the aggregation result. Only members with work to do participate: the same nodes that contribute query values (the leaf nodes) are also used as inner nodes in the aggregation tree.

The QueryKey<KT> object plays several roles here: it uniquely identifies the query (enabling Ida to match the result with the query request), is used in "trimming" these trees so that only nodes actually involved in the task need to play an active role, and also tracks the node that will be the root of the aggregation tree. This aggregation infrastructure combines the QueryKey data with the current membership view to build a spanning tree that only includes the nodes where the query actually has work to do. The QueryKey also incorporates the load-balancing feature mentioned above: for any given query, just one member of the relevant shards will participate, but the choice of member is varied to spread the work evenly.

There are three ways to identify the destinations for a message or a query. The first involves the entire group and maps to the multicast and Query APIs in Isis2. Such a message would reach user-provided handlers in all the members; the entire DHT would thus be searched. A second option is to list a desired ***subset*** of the group members, but otherwise "looks" similar. In this case, we employ the new Ida-introduced subset multicast. Finally, the third option involves specifying the destinations using a QueryKey. Here, the user provides a list of keys. Ida translates that list to a list of shards, then for each shard, picks a single representative in a manner that will balance loads.

As noted, while handling a query, components can use Put, much as a MapReduce operation triggers a shuffle during which results are moved about and aggregated for a Reduce operation. Indeed, one could issue a OrderedSend rather than a OrderedQuery, and just leave the participants to compute their partial results and then insert them into the DHT. This, though, raises some fault-tolerance issues; we'll discuss them in Section 5.

A final element of the API concerns control over garbage collection of key-value tuples. Again, there are several possibilities. First, Ida allows the user to specify a time-to-live value for the DHT as a whole. If this is used, any key-value pair is timestamped on creation and then automatically deleted after the TTL expires. The second option is more explicit: the user can Put "null" values, which will remove any prior key-value pair with a matching key. The third is simple but extreme: the group hosting the DHT can be closed, in which case the entire DHT will vanish.

5 Ida's Consistency Guarantees

The Ida consistency properties are as follows:

- OrderedPut operations are totally ordered with respect to one-another and with respect to OrderedGet and OrderedQuery.
- All non-faulty shard members apply each Put or OrderedPut, and in the same membership view.
- Key-Value pairs will not be lost unless a shard abruptly loses its full membership. After a join or leave (failure) event, data is automatically shuffled to reestablish the key-to-shard mapping, and any new shard members are correctly initialized with the full set of key-value pairs that map to them.
- The result of a query reflects exactly one contribution from each shard. Exceptions can occur either as a consequence of a failure, or because a shard referenced by the query has

become depopulated (*partial amnesia)*. Failures of this kind are rare, and can be handed by catching the exception and reissuing the query.

Because Ida was implemented over Isis2, Ida inherits a strongly consistent overarching execution model: a dynamically reconfigurable state machine architecture integrated with the virtual synchrony group communication model [3][2]. For our purposes the important guarantee is this: all group members see the same sequence of membership views (rank-ordered lists of members), and if a failure occurs, multicasts from the failed member are finalized before the view reporting the event is delivered. Thus the consistency properties listed above should be understood as being specified with respect to a particular view. The query exceptions mentioned in the fourth bullet, for example, arise when a new view is reported by Isis2 while a query is mid-way through execution. In such a case we can't guarantee our "one contribution per tuple" property, so we throw an exception and leave it to the application to reissue the query if desired.

Failure handling is greatly simplified by the underlying virtual synchrony model, and the primary-partition progress policy. Any multicast is either performed atomically (reaching all destinations that don't crash), or not performed at all, and new group membership upcalls are totally ordered with respect to multicast. Multi-Put and Multi-Get are thus failure atomic. Queries either return the correct response, reflecting exactly-once contributions from each participant, or an exception is thrown, enabling the application to reissue the operation.

A more complex situation arises if a failure disrupts an operation that generates new key-value pairs during the query step. Suppose that a query is being executed by just one member per shard, but one of those members fails. Not only would the query step fail, but if that member has sole responsibility for putting some subset of new tuples into the DHT, a new view will be defined, yet those tuples will be missing. Accordingly, applications code that create intermediate values must do so redundantly, by having all the shard members compute the new key-value pairs, and having all of them redundantly perform the needed put operations. DHT put collisions will thus be triggered. The Ida default behavior is to ignore duplicates. An overload of the QueryKey constructor allows the developer to explicitly request this behavior, and if these defaults are acceptable, no further action by the user is needed.

If a correlated failure causes an entire shard to become depopulated, Ida throws a shard-depopulation exception that would normally cause the whole DHT to shut down and then restart from scratch. Depopulation of a shard isn't a recoverable fault in our model because Ida state exists only in-memory. Fortunately, many modern cloud platforms allow long-running application to coordinate with the cloud management layer to ensure that elasticity won't cause correlated failures. By using these options and making shards large enough, such events can normally be avoided.

6 Implementation

Although the features of Isis2 simplified our task, Ida needs way to issue ordered reliable multicasts to subsets of a group. Isis2 lacked this form of subset multicast, nor can such a protocol be found in the literature (most reliable multicast protocols send messages within full groups that have membership that is either defined at the outset, or is built up over time through a series of join and leave events). The aggregation infrastructure is also unique, and again required new protocols. Finally, Ida repairs the DHT in an unusual way when members fail. We discuss all of these new mechanisms below.

Our discussion makes use of a slightly technical feature of the way Isis2 implements view changes. Isis2 membership change events occur in two stages: (1) an upcall occurs in all group members, warning them that a new view will soon be defined and giving advance notice of what that view will consist of (for example, lists of members that will be shown as having failed or joined). This permits members to terminate any pending protocols in a protocol-specific manner; we'll see two examples below. (2) the group membership service reports the new membership view, via upcall in all the members. Ida makes use of these mechanisms to terminate instances of our aggregation protocols and subset multicast protocols so that, as each new view becomes defined, the Ida layer can finalize any work initiated during the prior view.

In what follows, we ignore such issues as deciding whether to run over UDP or TCP, fragmenting very large messages into smaller chunks if needed, flow-control, etc. All are automated by Isis2 and we made no changes to these aspects of the system. Further, we won't say very much about large groups. When launched, Ida initializes a virtually synchronous group containing the members of the DHT, using a system call that allows a leader to specify the full membership (later, failures will trigger automatic removal of members from time to time, and planned elasticity events will be handled by the leader, which can add or remove members in batches). Thus even a large group can be created in a single atomic step. While replicating full membership may ultimately limit scalability, the issue has not yet been a problem.

6.1. Mapping Members and Keys to Shards

Ida group members share a consistent view of the group (consisting of the group name, a list of the members, and a view-id counter that increments by 1

each time the view changes). We used the ranking of members to determine the shard mapping. Members are ranked left to right, 0 to N-1. Ida computes the expected number of shards *(NS)* as the target group size divided by the shard size, then assigns the member with rank *r* to shard *r mod NS*, as was seen in Figure 3.

Given key K, GetHashCode(K) *mod NS* maps to the shard responsible for K. A QueryKey contains a list of keys, each separately mapped in this manner, resulting in a shard list that can be (much) smaller than the list of keys, since multiple keys might map to the same shard. A QueryKey also has a unique identifier and provides the rank within the group of the initiator: the root of the aggregation tree at which AggregationWait will occur, and where we want the aggregated result of the query to be available. Notice that by overriding GetHashCode, a developer can control the mapping of keys to shards.

Ida DHT groups often include more than their minimal number of members. These extra members play a special role if a failure occurs. Suppose a low-ranked member of the red shard fails (Figure 4, top). With a standard virtual synchrony scheme, a new view would be reported, and because all ranks will now have shifted down by 1, all members would be reassigned to new shards. A disruptive churn episode would ensue as data is shifted to the new representatives. Ida avoids this by hot-swapping processes from the end of the list into the gap: the member that previously had rank 19 in our example will be slotted into slot 5, and only it needs to be initialized (Figure 4, bottom). In this example the hot-swapped member was previously in the red shard, hence no data is copied, but in general a state transfer would then occur to initialize the repositioned process so that it can play its new role.

6.2. Implementing Put and Get

6.2.1. Best Effort Version

To establish a fair performance baseline, we implemented the Ida API in two ways. Our baseline version uses a standard, best-effort DHT architecture, with no special consistency properties: Put and Get operations map to a series of reliable 1-to-1 IPC operations, with Put sending the (key-value) pair directly to shard members where the key resides, and Get mapping each key to a shard member and then fetching the associated value using RPC.

With weakly consistent Put, a failure could disrupt the list of sends, so we needed an *eventual* consistency mechanism: a means of eventually repairing disruption caused by failures, but not necessarily doing so instantaneously. A standard DHT would use some form of background protocol to resynchronize shard members. Our task is simpler: as noted above, an Isis2 group membership change is preceded by an opportunity to terminate pending actions, so we simply have the termination upcall trigger an exchange of key-value lists (with large value objects, one can do this in two steps: sending key-version pairs, and then following up with an exchange of actual key-value pairs only to the extent needed). In this manner we obtain a reliable "1-hop" DHT, but without ordering guarantees: a Query could "overlap" with a Put, resulting in user-visible inconsistencies.

6.2.2. Strongly Consistent Version

We implemented our strongly consistent Ida OrderedPut and OrderedQuery operations by mapping them to *subset multicast*, a new protocol that we added to the Isis2 infrastructure. Given a QueryKey, we first map the keys to a set of shards. This list can then be mapped to a set of *participants* using the current group membership. We now run a protocol that uses an idea adapted from an early paper of Lamport's [19], which (to our knowledge) had previously been used only in full groups. The protocol assumes that all members maintain a logical clock (a long integer):

- The initiator sends a Put, Get or Query command *c* to each of the participants $\{P_0, \ldots P_k\}$.
- When *c* arrives at participant P_i, P_i increments its logical lock LT_i and sets $LT_c = LT_i$. P_i retains *c* in a *pending commands* record, and then sends LT_c back to the initiator.
- The initiator collects proposed times and the corresponding participant ranks, and then sends (LT_{max}, r_{max}) to the participants as a *commit time*. Here, LT_{max} is the maximum logical clock time within the set of proposed times, and the rank r_{max} (the corresponding rank) is used to break ties.
- Upon learning (LT_{max}, r_{max}), P_i updates $LT_i = \max(LT_i, LT_{max})$. Then, P_i updates the logical time of command *c*, setting it to (LT_{max}, r_{max}). P_i can execute committed command *c* when *c* has the smallest value of (LT_c,r) among the set of known commands (including both pending and committed commands. A committed command is delivered when are no prior pending commands on the queue, e.g. no c' with a smaller value of $(LT_{c'}, r')$.

It is easy to see that if commands c and c' overlap at participants P_i and P_j , then P_i and P_j will deliver c and c' in the order determined by the assigned commit times. Moreover, if P_i *later* learns of some command x, then because P_i uses its logical clock LT_i to propose an execution time, x will receive a commit time larger than the one used for c and c'.

From these properties we conclude that Ida operations occur along consistent cuts [8]: a Query or Get will be totally ordered with respect to Put. The protocol has *local costs*: only the initiator and the shards referenced in an operation participate. Slow processing of an operation by some single member

won't delay operations at other DHT members: the ordering protocol runs on a distinct thread from the one used to deliver upcalls to the application[1].

Should a failure occur, Isis2 notifies Ida prior to reporting the new view. Four cases arise**: (I-1)** the initiator may have failed after sending the command but prior to sending out the commit times; **(I-2)** the initiator may have failed after sending some commit messages but before all were successfully transmitted. **(P-1)** a participant may have failed before sending its proposed time, **(P-2)** a participant could fail after sending its proposed time but before performing the command (obviously, some combinations can arise).

[I-1] Ida takes no flush action in this case. When the new view is reported, Ida discards any pending commands initiated by a failed initiator.

[I-2] In this case, participants react to the flush upcall by echoing the commit messages they were sent by the failed initiator, via direct pt-to-pt messages to the other participants. The flush protocol forms a consistent cut in such a manner as to ensure that these will have been delivered and processed before a new view event could be delivered. Thus if *any* participant knows the commit time, the command will be executed by *every* non-failed participant. If none knows the time, then all garbage collect the associated pending messages from the queue, allowing pending committed messages with larger timestamps to be delivered.

[P-1] Here, a participant doesn't respond to the first phase message, leaving the initiator waiting. Finally a timeout occurs and the failure will be sensed. As long as the initiator is in the primary partition, Isis2 initiates a new-view flush upcall, and the initiator learns that the participant in question has failed. It can now terminate the protocol, and in fact can choose to commit or to abort the interrupted multicast (our version commits, using the subset of proposed times known so far). Should the initiator itself fail, a second view will be reported by Isis2 and a second flush protocol will run.

[P-2] If a participant fails while commit messages are being sent, its state is erased. A new view will be reported by Isis2 signaling the failure, and this will trigger repair, with some other node swapped in to replace the failed one. As for the failed process itself, no special action is needed.

This scheme assumes that commit information is retained until a new view is installed. Ida employs a concise representation of commit times to keep this data small, and garbage-collects the information in a lazy manner when it will longer be needed. We reset logical timestamps to 0 when a new view is installed.

6.3. QueryKey Shard-Member Selection

As noted earlier, an Ida Query only needs to be executed by a single participant per shard. For this case, the query initiator computes a hash over the QueryKey, and takes the result modulo the shard size. We then use this as an offset into the shard. Only the selected members participate in the query.

6.4. Aggregation Trees

Ida implements aggregation trees as overlays constructed over the set of participating members as leaves. Again, we leverage the guarantee that all group members use the identical membership view of the group, this time to ensure that all employ identical aggregation trees for any given query. We'll describe the handling for a single query, but Ida actually allows many to execute in parallel, using the QueryKey as an identifier to match each result to its query.

The aggregation mechanism is as follows. To initiate a query, we'll use ordered subset multicast to send it to participants. These participants form the leaves of our aggregation tree, and each performs a computation on the local set of key-value pairs, resulting in a *contribution* to the desired aggregate. These must now be combined to obtain the result of the query, as shown in Figure 5.

We define a binary tree by ordering the participants and numbering them 0… K-1. For simplicity, assume K is a power of 2. Call this layer 0 of the tree: the leaf set. The even numbered nodes now play aggregation roles: after node 1 computes its contribution, it sends it to node 0, which aggregates that value with its own value. In contrast, the other odd-numbered nodes compute their contribution and send them to their even-numbered counterparts. If a tree has an odd number of leaves, the even-numbered process at the end of the line can see this, and won't wait for a contribution. Having aggregated the layer 0 data, we treat the even-numbered nodes as the leaves of a new layer-1 tree, again numbering them from 0…K/2. Now the process repeats. At the end of the computation, node 0 will hold the aggregated result of the query.

To ensure that node 0 is in fact the initiator of the query, we simply set the initiator as the 0-ranked member of the participant list when mapping the key list to a participant list. The initiator thus participates as a layer-0 leaf whether or not any key mapped to the shard to which it belongs.

In the event of a failure, Ida interrupts pending aggregation query operations. A query initiator awaiting a result will receive an exception and can reissue the query in the new view. In this manner, we ensure that either an aggregation is successful, and

[1] Isis2 uses a single thread per group for event notifications, hence actions occur in the order that delivery is scheduled. If an application performs a very long-running action, we recommend that a new thread be spawned to avoid delaying subsequent actions. These threads must respect the read-write ordering implied by the event delivery ordering.

reflects exactly one contribution from each key, or that it fails. Combining this property with the consistent cut guarantee from the subset multicast used to initiate the query, we obtain the desired consistency guarantee.

6.5. Managing Intermediary Results

One interesting challenge posed by the Ida model concerns management of intermediate results formed during a multistage computation, and needed only for a brief period of time. There are several options. First, each Ida tuple has a TTL field; if this is set, the tuple will silently vanish when the TTL expires. Next, there is an explicit deletion option for garbage collection under user control. And finally, one can create an entire temporary group, populate it with temporary data, and then delete the whole group when finished. This form of deletion is very efficient and particularly easy to implement.

7 Use cases

We believe that most uses of Ida will be fairly simple (such as finding the value associated with key "K", looking up keys in key-value pairs with value "V", counting items that satisfy some pattern either on keys or on values, finding the top N items under some ranking, etc). It is trivial to code these kinds of actions, and the ability to update or query multiple key-value pairs in a single atomic action eliminates what might otherwise be complex logic.

Less clear is the question of whether users would want to run more complex applications on Ida. The puzzle is as follows: all sorts of complex data structures can be mapped into a key-value representation, queried, and updated, and hence one can map some remarkably sophisticated applications into Ida. However, the resulting algorithms often require multiple steps to carry out a single action. One thus is forced to ask whether the solution departs from the DHT properties that make a system like Ida appealing. Our belief is that while Ida can handle a wide range of what might normally be viewed as transactional database applications, relatively few users will try to do such things, and they will mostly limit themselves to cases that require very few Ida actions.

For example, with Ida it is very easy to use an iterated binary search to find the median of a set of values for a set of matching keys. Similarly, continuous tracking of an evolving aggregated property of the underlying data set can be done by writing code to compute the property and then periodically issuing initiation events. In both cases log(N) steps are required; our conjecture is that while such a cost would be too high if it incurred for all operations, occasional actions with this cost might be tolerable.

Also relatively easy to implement are multidimensional searches. There are a few ways to solve such problems; the easiest is as follows. We take the ({key-list},value) tuple and compute its hash code, using this as a primary key and storing the tuple at that location. Next, for each key in the key list, we store a "pointer" mapping from that key to the primary key (the UID if you prefer) for the object. It is helpful to define a type for each dimension, thereby leveraging the Ida type system as a way to distinguish the various dimensions of the key list. Now we can do a lookup on any subset of the keys. One issues a query, then aggregates the results, finding matches on each subkey first and retaining only the tuples that matched in all keys; a "join" encoded into the aggregation method.

This approach can also support multi-key *range* queries and *pattern* search: one sends the request to all shards where a match could arise, and then does an aggregation that will run in time log(K) where K is the number of shards touched: precisely the search cost seen in existing solutions. In the worst case, the query can simply search all shards.

The generalization of this line of thinking is as follows: since Ida permits the developer to perform a "one-shot" atomic action on the DHT, are one-shot transactions a general enough construct to support a wide range of what might normally be thought of as database queries and updates, or is the "all at once" aspect too limiting? As discussed in the introduction, any response to this question must carry a caveat: we are interested not just in understanding the extent to which a DHT can mimic a more complex database, but rather the degree to which a DHT can perform database operations while preserving the essential DHT-ness of the solution. For example, if we were to solve such a problem using locks (which Isis2 does support), or with a full transactional mechanism allowing multiple operations and ending with a commit or abort, we would simply be building a new kind of full-scale database and abandoning the properties that make a DHT appealing.

Although brevity precludes a detailed discussion of such algorithms, we've looked at this model and can summarize our findings. First, we've identified a number of ways of "flattening" transactions, so that a transaction designed with multiple stages could potentially be transformed into a small number of highly parallel steps. To do this, one draws on an old idea: by pre-executing a transaction on a read-only copy of a database, it is possible to compute the read and write set that would be used if we ran the transaction "now". For example, transaction T might read version 7 of X and version 9 of Y, then update Z, creating version 3 of Z and setting it to 123. The updates are captured but not applied: they form an *intentions list*.

Of course, with dynamic updates, any object can be updated at any time. Accordingly, having flattened the

transaction in the first stage, one resubmits the modified form of the transaction to run as a true update action in a second phase. Specifically, now that we know the intended read and write operations, we can skip the computation! Instead, the requirement is simply to verify that the versions of X, Y and Z are the same as they were during pre-execution, then update Z. More broadly, the rule is to verify the version numbers on all objects that will be read or written, then atomically apply all the updates in the intentions list if the verification is successful, discarding the updates and restating the whole process if not. Clearly, this is a win if aborts would be rare, and could be a terrible idea in a high contention situation.

Ida lends itself nicely to this form of computation. In the pre-execution phase, the mini-transaction is run like any Ida query. We use aggregation to construct a list of the object versions that were touched and form the intentions list. Now in a second Ida operation, we re-issue the transaction as a "proposed" update. Recipients log the portion of the intentions list corresponding to keys that map to their shard, verify object versions (temporarily locking those objects), and then respond with a commit or abort vote, retaining the proposed updates. Again, we aggregate, and then send out a commit or abort outcome multicast. Since all multicasts in Isis2 and Ida are atomic with respect to group membership changes, if a failure disrupts such a sequence, participants will see a new group view in which the transaction initiator has failed. They can simply discard the proposed updates: no participant could have received a commit, because if any had, all would have, and they would have seen it prior to the reporting of the new view.

Thus Ida can support fairly elaborate transactional behaviors that are carried out in a three-phase process that uses (at least) two Ida aggregation queries and one multicast. During the period from when the second phase runs to when the third phase finishes, participants retain but can't yet apply updates and may need to delay other reads or writes on the affected objects. This takes us full circle: are we now so far from the DHT model that users wouldn't consider Ida as a suitable platform for this style of computation? While the example shows that Ida has computational power equivalent to many kinds of databases, without a serious side-by-side comparison, we can't easily guess at how our performance would stack up.

8 Analysis and Evaluation

Our experiments focus on the incremental cost associated with consistency and on establishing locality of costs. To this end, we compare consistent and non-consistent Put, Get and Query, all running on a dedicated cluster (future experiments will be carried out on much larger platforms). Our main goal is to show that costs are local and that Ida scales well and rapidly recovers from failures.

We see the results of these experiments in Figures 6-8. All were conducted on a pair of very similar HPC clusters, one at Cornell; the other at LLNL labs. We reserved sets of dedicated nodes, then ran either a single copy of our test program on each node, or 4 copies each. The nodes themselves are dual Intel Xenon 6-core processors running at 2.8GHz, and interconnected via Mellanox InfiniBand switches. The UDP message size limit for these connections is 64K, and the Infiniband MTU is 4K. Each node has 48GB RAM memory. The clusters both run identical versions of Linux. Our code was compiled using the 3.2.1 version of the Mono framework.

Before presenting our data, we need to remark on the poor performance of Infiniband when used to support point to point UDP and IP multicast [18]. As the reader may be aware, Infiniband is capable of exceptionally low latencies and high data transfer rates, and can even support a hardware mediated form of direct access to remote memory, as well as DMA copying from a sender directly to a receiver. However, to gain these benefits one must either use Infiniband directly through a so-called "verbs" API, or focus on very large data transfers. In today's clusters, Infiniband is often configured to emulate an Ethernet, offering UDP, TCP and IP multicast functionality. Studies have shown that in this situation, Infiniband dominates 10G Ethernet in almost all respects, but that the Infiniband UDP protocol is surprisingly slow, achieving less than half the message rate of 10G Ethernet. Because Ida is a UDP-based system, our experiments turned out to stress this weak point in the IB performance space. We had no choice: the large clusters available to us both require that messaging-intensive experiments run on the IB network. In the future we hope to repeat our experiments both using the Infiniband verbs API, and on 10G Ethernet.

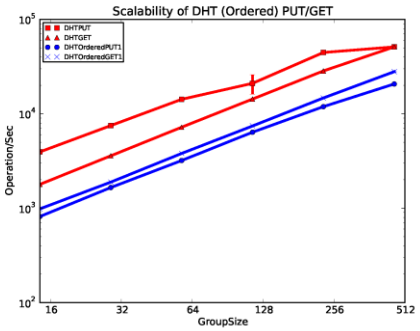

Figure 6: Total capacity of a DHT for Put/Get versus OrderedPut/OrderedGet. Nodes initiate operations at the maximum sustainable rate.

The experiments are symmetric: except as indicated, all nodes perform identical operations, with different keys. The shard size was two: a group of size N has N/2 shards. For Figure 6, we ran 1 copy of our Ida test application per node and allocated 2 cores each. Each experiment ran for roughly 5 minutes of wall-clock time (the number of operations depends on the scenario, but was in the tens or hundreds of thousands). Figures 7-10 ran 4 copies per node, again with 2 cores per copy; performance and loads were unchanged from the 1-instance per node run. Performance figures are for full runs; the error bars are across runs.

By construction, the Ida operations have expected O(1) cost for eventually-consistent Put and Get operations: the DHT itself is "one-hop" because each node has the addresses of all the other nodes available to it, and can perform operations that access multiple shards in a single parallel step; cleanup after a failure is a rare event and just involves pushing updates to nodes that may lack them. OrderedPut and OrderedGet use the 2-phase protocol of Section 6.2.2 for the initial step, with several consequences: whereas Put can run in an asynchronous "pipelined" manner, OrderedPut will be delayed by this initial step. Further, while Put and OrderedPut must touch every member of any shards the keys map to; Get and OrderedGet only interact with one member per shard.

Figure 7 explores the total performance achieved for groups of various sizes using the standard DHT Put and Get side by side with our ordered versions, evaluating single-key operations, employing a shard size of 3 and value sizes of 100 bytes (the size had little impact on performance; Ida performance turns out to be nearly the identical for a wide range of value sizes from 10 bytes to more than 62K). Variance was low.

We see linear scalability, but the raw numbers are lower than one might expect, apparently because of the sluggish Infiniband UDP implementation mentioned earlier. To arrive at this conclusion, we first explored a number of possible limiting performance factors. In this experiment all requests were initiated by a single thread; with multi-threaded initiators Ida achieves higher performance. Space limits precluded detailed discussion of that option here, but the benefit is at most 10%. Notice that OrderedGet and classic Get have similar but not identical performance: the implementation of the OrderedSend protocol employs some synchronization that can be avoided by the classic version, but it is revealing that the very simple Get protocol isn't very much faster. Recall that this version of Get maps directly to an RPC. Thus if it runs

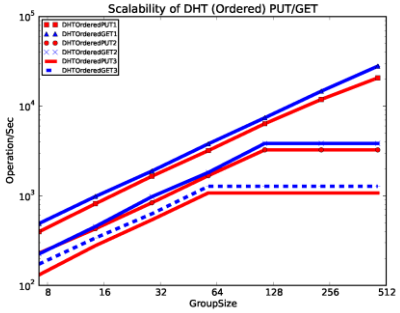

Figure 7: Ordered DHT operations with varying numbers of keys (1, 2 or 3). The Infiniband NIC becomes a bottleneck as this test scales up.

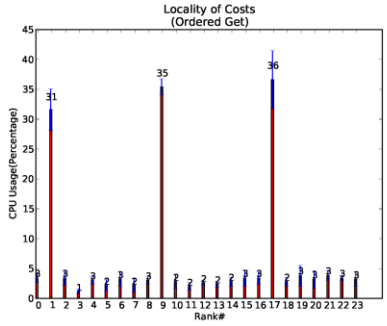

Figure 8: CPU usage for OrderedGet with a fixed set of keys. The load is highly localized.

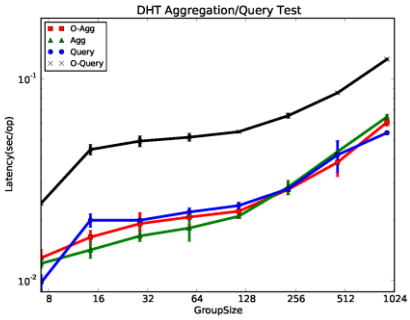

Figure 9: Queries done in two ways: using Query/OrderedQuery (initiator collects replies) and then with Aggregation started by Send/OrderedSend.

Figure 10: While a stream of OrderedGet requests are performed, the DHT is forced to recover from failures at times 30 and 60.

slowly, attention focuses on flow control (we checked and it never kicks in for these experiments), O/S overheads, and the Infiniband version of UDP.

We ruled out any form of CPU bottleneck: in our experiments, CPU loads (including both application and O/S computational overheads) never exceeded 30-40% (for example, see the discussion of Figure 8), and profiles showed that Ida itself spends most its time in the .NET code that waits for incoming data and then retrieves the message from sockets. It follows that Infiniband limits performance. To confirm this conclusion we ran experiments in smaller clusters with switched Ethernet interfaces, and consistently achieved much higher messaging rates.

In Figure 7 we measure performance for a single thread issuing a series of OrderedGet requests, with varying numbers of keys (each key to a distinct shard, and those shards are never the one to which the initiator belongs). Recall that we only need one participant per shard, hence with a single key (a single participant), OrderedGet becomes a classical Get, whereas with two or more keys it runs the ordered subset multicast protocol of Sec 6.2.2. Notice that the network link reaches peak load and becomes a bottleneck in the 2-key and 3-key experiments.

Figure 8 explores CPU usage when OrderedGet requests are issued using a key pattern designed to select just certain shards. We see that costs are extremely localized: only the participating nodes incur load. This is good for scalability if work can be spread evenly, but as just noted, can also create hot-spots that would be points of contention. Further experiments at larger scale will be needed to clarify the tradeoff. Notice also that at peak load these nodes were only at 30%-35% CPU utilization, supporting the view that the network is our bottleneck.

Figure 9 compares two query scenarios, one using aggregation (Send/OrderedSend) and the other using direct all-to-one replies (Query/OrderedQuery), with the "O" in the figures designating strongly consistent versions. A single initiator is picked and issues a stream of queries, holding the number of participants constant but cycling through a key pattern designed to eventually touch the full group, with 8 participants per request. Latencies are low here, but ordered aggregation has a visible delay. As the group grows, the gap shrinks. With larger reply objects or very large numbers of participants, we would start to see packet loss at the initiator for the direct replies case, and aggregation would certainly win, but this experiment apparently didn't trigger loss.

Finally, in Figure 10 we measured the speed of group reconfiguration. We issued a stream of Get operations in a group of size 55. After 30 seconds the member ranked 5 terminates. We see that this causes a brief perturbation in the rate of Get operations that can be sustained. At time 60, members 19-37 terminate. Again, recovery is quick, but this time when the group reconfigures, it has lost 1/3 of its computing capacity.

9 Prior work

While there has been extensive prior work with DHTs, fewer efforts have explored consistency when updates are mixed with queries. Notable among these are Dynamo [12], Pastry [7], Scatter [15], Spark [31], Naiad [24], and Comet [14]. Although none of these has the style of expanded DHT API used in Ida, Comet does explore various extensions aimed at customizing the DHT. The motivation in that project was actually similar to ours, but Comet only offers consistency on a single-tuple-at-a-time basis, hence many of the issues discussed in our paper didn't arise in their work.

Spark is a well-known in-memory cache with interesting similarities to Ida. Unlike Ida however, Spark doesn't replicate data or move it from node-to-node; any given Spark instance is basically a local cache on the node where it runs. The design goals differ too: Spark helps iterated MapReduce computations avoid recomputing partial results, using LINQ-like queries (coded in Scala) as keys to byte vectors representing serialized intermediate results, which are retained for future reuse, and implementing a novel scheduler that places MapReduce tasks so as to maximize reuse potential. Moreover, Spark basically assumes an immutable data store: it supports the creation of intermediary results and has an extensive infrastructure to optimize the management of that form of data, but updates to the underlying data set occur only through append-only actions that create a series of databases, each corresponding to the "time" it represents. A computation runs in just one of these snapshots [31].

Ida could be used in much the same way, but our emphasis is on general updates to a key-value set, not on append-only big-data stores; we replicate data within shards, and whereas Spark can easily recompute data as needed, Ida treats key-value objects as information that can't casually be discarded. On the other hand, Ida offers no recourse if a crash depopulates a DHT or a shard; Spark potentially can recover lost data from a backing store and can recover lost intermediary results by recomputing them.

The desire to offer strong consistency for multi-tuple updates and queries motivated a key contribution in Ida: whereas prior work [15] used Paxos [20] for consistency and durability on a shard-by-shard basis, Paxos cannot be used for varying sets of shards, since this gives rise to irregular patterns of membership overlap. Furthermore, because Ida runs in the soft-state tier of the cloud, durability of the kind offered by Paxos isn't needed. Thus Ida's multi-tuple API requires an ordered subset multicast, and by solving

this problem, it becomes possible to guarantee consistency for collections of tuples and for queries that might span many shards.

We pointed out that both Ida and Spark leverage the LINQ query language. The use of LINQ in this manner is natural and traces back to Dryad/LINQ [17] and Naiad [24], which pioneered the use of LINQ as a computational API for operations on key-value collections. LINQ makes these systems somewhat database-like, and indeed in Section 7 we saw that with a bit of effort Ida could be used as a full-featured in-memory database system. However, doing seems to be at odds with DHT-ness. As a result, our focus is on uses that wouldn't guarantee full ACID properties: we favor "one-shot" atomicity, and work primarily with in-memory data, yielding weak durability.

Because Ida can be used for queries across large collections of tuples, we should touch upon prior work on DHTs such as rKelips [21] and HyperDex [13]. These key-value systems both support range queries. rKelips, like Ida, uses a 1-hop DHT architecture but only allows range queries in one dimension. HyperDex employs a log(N) structure but allows range-queries in multiple dimensions. One carries out range-queries in Ida by issuing a query that spans the DHT shards that could include key-value pairs of interest and then carrying an appropriate computation on the collection, aggregating the results: this last step would, similarly, take log(N) time. Notice, however, that because it may be hard to know which key-value pairs might match, in practice HyperDex would often be more efficient for this purpose: the Ida query may need to access shards where there are actually no matching tuples.

As noted in the paper, Ida exists in a space that also includes full transactional databases: Google's Spanner [10] was mentioned several times, and one could also point to systems like Picolo [25], which optimizes for storage of graphical models. We believe that because these depart from the simple and predictable costs that make DHTs so popular, they wouldn't represent appealing options for the community we target. Ida, by retaining the DHT performance properties while enhancing consistency, aims at developers who understand and value the DHT model but who need an enhanced API and stronger properties.

Our examples touched upon simple graph algorithms, since these illustrate both the potential power of a consistent key-value store that accepts updates, but also the more complex programming style required when working with such a store in a lock-free manner. As noted, we're doubtful that users will favor Ida for the most complex such solutions, even if there is a way to map complex structures and complicated transactions into the Ida model: doing so departs from the DHT-ness of Ida, which we believe will be key to adoption. Thus a developer facing such problems might well prefer optimized systems specifically aimed at graph processing at scale, such as Google's Pregel system [22]. Microsoft's Naiad platform is a second example in this class; it focuses on fix-point computations in very large graphs [24]. Twitter's Storm [23] streaming query system can query high-rate streams of key-value pairs; this is a slightly different problem than the one we consider.

Ida's aggregation scheme is reminiscent of Astrolabe [27]. However, that system lacked strong consistency, whereas Ida has very strong guarantees. Moreover, Astrolabe imposed a strict size-bound on query results; Ida aggregation allows unbounded objects to be accumulated, if desired.

10 Conclusions

Ida is an in-memory DHT offering a substantially expanded API capable of supporting a wide range of data structures, with guarantees of strong consistency even when updates and queries concurrently modify large numbers of key-value pairs. The system is lightweight and highly scalable, offering the guarantee that each query reflects exactly one contribution from each relevant key-value pair.

11 Acknowledgements

The authors are very grateful to Greg Bronevetsky, Al Demers, Michael Isard and Robbert van Renesse, and the TRIOS and SOSP PC members. We were funded by DARPA and NSF. The experiments reported here were performed on clusters provided by LLNL, and on Cornell's NSF-funded Atlas cluster.

12 References

[1] K.P. Birman. isis2.codeplex.com

[2] K.P. Birman, T.A. Joseph. Reliable communication in presence of failures. Kenneth ACM Trans. on Computer Systems (TOCS), Vol. 5, No. 1, Feb. 1987

[3] K.P. Birman, D. Malkhi, R. van Renesse. Virtually Synchronous Methodology for Dynamic Service Replication. Appears as Appendix A in [4].

[4] K. Birman. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services.* 2012, Springer-Verlag.

[5] K. P. Birman, D. Freedman, Q. Huang and P. Dowell. Overcoming CAP with Consistent Soft-State Replication.. IEEE Computer Magazine (special issue on "The Growing Impact of the CAP Theorem"). Volume 12. pp. 50-58. February 2012.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. OSDI 2006.

[7] M. Castro, M. Costa, and A. Rowstron. Performance and Dependability of Structured Peer-to-Peer overlays. In Proc. Of DSN, 2004.

[8] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, Feb 1985.

[9] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, I. Stoica. Managing data transfers in computer clusters with orchestra. ACM SIGCOMM 2011.

[10] J.C. Corbett, *et al.* Spanner: Google's Globally-Distributed Database. OSDI 2012.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.

[12] G. DeCandia, G., *et. al*. Dynamo: Amazon's highly available key-value store. In Proceedings of the 21st ACM SOSP (Stevenson, Washington, October 2007).

[13] R. Escriva, B. Wong, G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store. ACM SIGCOMM '12, Helsinki, Finland, 2012

[14] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, Henry M. Levy. Comet: An active distributed key-value store. OSDI 2010.

[15] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, T. Anderson. Scalable Consistency in Scatter. ACM SOSP '13, December 2011, Cascais Portugal.

[16] J. Gonzalez, Y. Low, H. Gu,, D. Bickson, C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012.

[17] M Isard, M Budiu, Y Yu, A Birrell, and D Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007).

[18] S. Kandadai and X. He. Performance of HPC Applications over InfiniBand, 10 Gb and 1 Gb Ethernet. IBM Corp., Sept 2010 WhitePaper. http://www.chelsio.com/assetlibrary/whitepapers/HPC-APPS-PERF-IBM.pdf

[19] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, ACM Trans. on Computer Systems 6(2): 1984, 254-280.

[20] L. Lamport. The Part-Time Parliament. ACM Trans. on Computer Systems 16 (2): 1996.

[21] P. Linga. Indexing in Peer-To-Peer Systems. Cornell Univ. Dept. of CS. (http://www.cs.cornell.edu/projects/quicksilver/public_pdfs/prakash-thesis.pdf). Doctoral Thesis. May 2007.

[22] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 135-146.

[23] N. Martz. A Storm is coming: Details and plans for [Twitter's first Storm] release. Aug. 2011. http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html.

[24] F. McSherry, R. Isaacs, M. Isard, and D. Murray, Composable Incremental and Iterative Data-Parallel Computation with Naiad, no. MSR-TR-2012-105, 9 October 2012

[25] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. OSDI 2010.

[26] F. Schintke, A. Reinefeld, S. Haridi, T. Schütt: Enhanced Paxos Commit for Transactions on DHTs. CCGRID 2010: 448-454.

[27] R. van Renesse, K. Birman and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. ACM TOCS 21:2, May 2003.

[28] R. van Renesse and H. Johansen. Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. EuroSys 2006.

[29] Y. Vigfusson, *et. al.* Dr. Multicast: Rx for Data Center Communication Scalability. Eurosys 2010.

[30] Y. Yu, M. Isard, *et. al*. Dryad/LINQ: a system for general-purpose distributed data-parallel computing using a high-level language. Proc. OSDI'08 Dec. 2008

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012, April 2012.