SPICE: IMPLICIT GROUP MESSAGING ON ICE
TECHNICAL REPORT 618

DANIEL CUTTING AND BJÖRN LANDFELDT
SCHOOL OF INFORMATION TECHNOLOGIES
THE UNIVERSITY OF SYDNEY

AARON QUIGLEY
SCHOOL OF COMPUTER SCIENCE & INFORMATICS
UNIVERSITY COLLEGE DUBLIN

AUGUST, 2007
Abstract

This technical report describes the SPICE implementation of IGM based on the ICE tesseral P2P substrate. SPICE employs advanced load distribution techniques—enabled by the unique features of ICE—to ensure fair loading of all peers and network links.

1 Introduction

Implicit group messaging (IGM) is a decoupled many-to-many paradigm for delivering messages from publishers to groups of consumers. Publishers select groups of consumers by their attributes rather than by name or address. Any participant is capable of publishing messages to any implicit groups at any time. The actual membership of the same implicit group may vary from message to message as participants join and leave the system or as their attributes change over time. IGM is described in detail in an associated technical report [2].

ICE is a novel P2P substrate based around tesseral addressing and an amortised multicast routing algorithm. It is a useful fundament for building P2P systems and has just two functions: to organise the peers into a structured overlay over the physical network; and to provide a routing algorithm for delivering messages from source peers to multiple destination peers. ICE is described in an associated report [3].

This report details the SPICE implementation of IGM using the ICE substrate. SPICE supports an IGM modelling language based on simple descriptive tagging, although instead of tagging content as is often the case, participants tag themselves. This language is presented as a concrete example in the IGM technical report [2].

2 Basic SPICE

The SPICE implementation of IGM delivers casts over an homogeneous network of peers that cooperatively serve consumers. A structured P2P substrate called ICE [3] provides a “surface” upon which the SPICE registration and casting algorithms are founded. Many “rendezvous points” on the surface are used to store tag registrations and marshal casts for delivery to consumers. Advanced distribution and replication techniques, described in Section 4, ensure the participating peers perform only limited work and store limited state information, independent of the number of peers, popularity of tags, or frequency of casts.

Many structured P2P designs exist; their common feature is an address space within which peers are arranged that permits deterministic routing. Collectively these are often referred to as Distributed Hash Tables (DHTs) or Distributed Object Location and Routing (DOLR) designs. A DHT conceptually operates like a classical hash table, storing and retrieving objects over the network via fixed length keys. Peers are typically responsible for a portion of the address space and store and serve the objects that are hashed to it. DOLRs are very similar but support routing of arbitrary messages to objects or nodes in the overlay, rather than just storage and lookups.

A simple DHT-based IGM system is easily imple-
mented in two parts. To register, each peer inserts a reference to itself at the hashed address of each of the tags in its registration, termed rendezvous points (RPs). To cast, a publisher retrieves across the DHT all peer references stored for each of the tags in the target expression. It then combines and intersects them according to the expression before notifying the resultant set of the message. Such an approach has much in common with distributed search indices for document collections which employ “vertical” or “keyword partitioning” [9, 14].

A problem with this naïve iterative approach is that considerable intermediate data must be returned to the publisher before it can determine the selected consumers. A recursive alternative is to chain the cast through several RPs, as applied in Panaché [6] and RDFPeers [1], for example. At each point, the intersection of results can be calculated and forwarded to the next RP, reducing the amount of data that needs to eventually return to the publisher.

However, the network traffic cost for lookups is non-negligible, whether iterative or recursive. The number of lookups can be reduced by each peer storing a complete list of its tags at the RP for each of its tags. A lookup then returns all tags for a peer and allows the set of selected consumers to be calculated without having to lookup every tag individually. A somewhat similar strategy is employed in keyword fusion [5] and Keyword-Set Search System [4].

The RP for each tag must be informed when a participant’s registration is altered, which may be a reasonably expensive operation in terms of network traffic. However, IGM registrations represent interests or inherent attributes of a participant which presumably will not change as quickly as casts are published. Since casts may occur frequently and to potentially very large groups, their efficiency is a priority. Thus SPICE is optimised for casts, and stores all of a peer’s tags at the RP for each.

Even with this improvement, results are still returned to the publisher before consumers are notified of a message. Since all the information necessary to find selected peers is stored at RPs, a simple optimisation is for the peer responsible for the RPs to forward casts directly to consumers instead, reducing the total amount of data transferred around the network. Note that as the selection logic is not centralised at the publisher, it is possible for duplicates to be delivered to consumers selected by multiple disjunctive terms of a cast. This issue is discussed and resolved in Section 3.2. Because other peers in the network must now participate in the actual delivery of casts, such an approach requires a DOLR (where peers execute specific algorithms upon receipt of messages) as opposed to a pure lookup-based DHT. This is the fundamental SPICE design, illustrated in Figures 1(a) and 1(b).

Listing 1 presents the basic SPICE design in object-oriented pseudocode resembling a high-level language called Ruby [12]. The reg and cast methods of the IGM interface [2] are implemented by client peers. The send method is assumed to transmit a message between two peers (over unicast). The notify method (from the IGM interface) is identical to send but is used to distinguish a cast’s final hop to a consumer. The route method routes a message over the DOLR to a hashed address. Method names prefixed with handle_X are invoked when a message of type X is routed to any peer in the DOLR (recall that peers are homogenous and thus any is capable of handling a message).

Table 1 defines the important fields and methods in the objects used by the algorithms. The or_terms field of the Target object is a collection of the disjunctive terms of the target expression. Other fields are explained in the listing as necessary, and in the following sections.
Listing 1 Algorithms for basic SPICE on a DOLR.

```python
def reg(p):
    # Route the registration to the RP for each tag.
    p.registration.tags.each do |tag|
        # RP peers may store several registries (due to hash collisions).
        # The tag field identifies the registry for this registration.
        p.registration.tag = tag
        p.route(tag.rp,p.registration.dup)
    end
end

def cast(p,c):
    c.target.or_terms.each do |term|
        # The term field indicates which disjunctive term the RP is
        # being asked to resolve. This is used to prevent duplicate
        # notifications (Section 3.2).
        c.term = term
        # Casts are handled by a single registry. Since some RPs hold
        # several registries, the tag field identifies the one to use.
        c.tag = term.first
        p.route(c.tag.rp,c)
    end
end

def handle_reg(p,r):
    # Add it to the appropriate registry.
    p.registry[r.tag].push(r)
end

def handle_cast(p,c):
    registrations = p.registry[c.tag].find.all do |r|
        c.target.selects?(r)
    end
    # minimum_term? prevents duplicate notifications (Section 3.2).
    registrations.each { |r| p.notify(r.peer,c) if minimum_term?(r,c) }
end
```

Table 1: Object structure for basic SPICE algorithms.

<table>
<thead>
<tr>
<th>Peer</th>
<th>registration</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) A peer object.</td>
<td>(b) A registration object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cast</th>
<th>target</th>
<th>term</th>
<th>tag</th>
<th>payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) A cast object.</td>
<td>(d) A target expression object.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
<th>or_terms</th>
<th>selects?(registration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e) A target expression object.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3 SPICE on ICE

This section describes how the basic SPICE algorithms are constructed on top of ICE. Note that for the remainder of the report, RP can refer both to a rendezvous point on the surface and the actual peer storing registries.

3.1 Registration

A peer routes its registration over the ICE surface to the RPs for each of its tags where it is stored in a registry. DHTs typically use consistent hashing to map objects to identifiers in the DHT address space [7, 11, 13, 8, 10]. The number of bits in the address space is often fixed at 128 (suitable for the MD5 hash function) or 160 (suitable for SHA-1).

Due to ICE’s tesseral addressing, the rendezvous points for tags are in fact deep extents and are interchangeably referred to as rendezvous extents. The address of a rendezvous extent is found by hashing a tag, then taking \(d\) bits at a time for each digit.

However, each registry in SPICE should be “owned” by exactly one peer, so the depth of the rendezvous extent must be sufficient to ensure that only one peer’s tract intersects it. The size of a peer’s tract is dependent upon the number of peers \(n\), the dimensionality of the surface \(d\), and the way the surface is partitioned. If the surface is partitioned equally, the number of digits needed for a rendezvous extent to map to a single peer is:

\[
digits = \left\lceil \log_2 n \right\rceil \frac{d}{d} (1)\]

For example, a 2-dimensional surface with 4 equally partitioned peers requires just one digit for a rendezvous extent to map a tag to a unique peer. Similarly, a 3-dimensional surface with 512 peers requires 3 digits. Since each digit is \(d\) bits, \(O(\log_2 n)\) bits are required for rendezvous extent addresses. Thus only 20 bits are required to support a million peers. Since the surface partitioning will not be completely equal in practice, a safe margin of extra digits is required to ensure that a rendezvous extent maps to a single peer. However, a hash function such as SHA-1 is capable of easily supporting any realistic number of peers. Note that as addresses need to be expressed in terms of digits of a particular surface dimensionality, the tail of the hash is dropped as necessary to ensure the number of bits is divisible by \(d\).

3.2 Casting

As outlined in Section 2, casting to implicit groups requires publishers to route messages to rendezvous points on the surface which then notify all selected consumers stored in their registries.

Listing 1 and Table 1 present the cast algorithms and supporting objects for the basic SPICE implementation. The cast’s tag field contains the particular registry for which the cast is destined. Because an RP may be responsible for many tags, this allows the RP to determine for which registry it should be processing the cast. The term field is similar, containing the disjunctive term for which the message should be processed. This is needed to avoid notifying consumers repeatedly of casts when they are selected by more than one disjunctive term (explained below).

In the case of a target expression of just a single tag, the publisher routes the cast to the rendezvous extent of that tag. Upon receipt, the RP iterates through all registrations stored for the tag and notifies consumers.

A target expression containing conjunctive terms (e.g., football&jazz) is handled very similarly. The publisher selects any one of the tags and forwards the cast to the RP for only that tag. Upon receipt, the RP searches the registrations for that tag, but only notifies those peers that have registered all tags in the target expression. This is possible because the registration contains the entire registration of the peers. Note that the single tag case is actually a degenerate version of this case.

Casting to an implicit group using disjunctive terms is slightly more complex. Consider the target expression samba|jazz. This cast should be delivered to all consumers that have registered samba, or jazz, or both. Therefore, using a single RP will not suffice, because it will not necessarily have enough information to find all of the selected consumers. For instance, if the message were routed to the RP for
Tests if a consumer should be notified of a cast based on the total ordering of terms, and whether the RP handling the cast is responsible for the minimum term.

```ruby
def minimum_term?(reg, cast)
  selected_by = cast.target.or_terms.find_all do |term|
    term.selects? reg
  end
  cast.term == selected_by.min { |x,y| x.sort.join <=> y.sort.join }
end
```

For both of these tags. By only testing the summary against the target expression `samba|jazz`, both RPs will determine that this consumer should receive a copy of the cast. To determine which should notify it, each RP finds the minimum of the two disjunctive terms, e.g., `jazz` (because `jazz < samba` lexicographically). The RP for `jazz` will therefore exclusively notify the consumer.

### 4 SPICE Unloaded

Due to its homogenous design, the basic SPICE implementation is sufficiently fair when peer registration and cast distributions are uniform. However, these distributions are more likely to follow Zipfian distributions of varying skew. In such conditions, the basic SPICE implementation suffers severe loading of RPs. This section therefore extends the basic design by making use of the novel features offered by the ICE surface: its hierarchical tesseral addressing and amortised routing algorithm.

Table 2 enumerates four extreme points on the continuum of possible casts and shows their effect on the incoming and outgoing loads of RPs. The work of any one peer should be minimised despite this variety of
loading possibilities. For small implicit groups, it is best to deliver casts directly from RPs to each member as this minimises delay without greatly affecting the network. For large groups, notification places a high outgoing load on RPs and the links surrounding them, so a better solution is to spread the notification load to other peers. Finally, RP peers for tags frequently appearing in casts may have a high incoming load. This incoming cost should be shared with other peers. These are the primary guiding principles of the load distribution techniques presented in this section.

Two new parameters are needed to capture these principles. The \textit{storage limit} (SL) is the maximum number of registrations a rendezvous peer is willing to store for a particular registry. The \textit{frequency limit} (FL) is the maximum frequency of casts (per second) a rendezvous peer is prepared to service. The intention is to reduce storage and outgoing load on RPs that hold registries for common tags, and reduce incoming load on RPs that hold registries for tags that appear frequently in target expressions.

The storage and frequency limits guide the operation of two complementary load distribution techniques called \textit{registry distribution} and \textit{registry replication}. The former adaptively distributes the registrations held in a registry over many peers surrounding the original RP, such that each is required to store and forward only a fraction of the total outgoing load for a cast. The latter makes complete copies of registries at other parts of the surface that can be independently used to resolve casts, thereby reducing the total incoming load on any single RP. These techniques can be combined for the registries of tags that are both commonly registered by peers and frequently used in target expressions.

### 4.1 Registry distribution

The basic SPICE registration algorithm stores registries at deep rendezvous extents on the ICE surface. When many peers register the same tags, these registries become large and the RPs responsible for them must store many registrations and notify many consumers.

Registry distribution is designed to reduce the storage and outgoing load of these RPs. It works by incrementally scaling the address of where a registry is stored from the initially deep rendezvous extent to the entire ICE surface, using the hierarchical property of the tesseral addressing scheme. The initial rendezvous extent found by hashing a tag can be thought of as the deepest possible container for all registrations. At this depth it is covered by a single peer which is solely responsible for the registry. When an RP reaches its storage limit, the rendezvous extent is scaled by omitting a digit from the end of its address. The new container is $2^d$ times the size and intersects with the tracts of more peers, each of which becomes an RP for that registry and is responsible for storing a fraction of the registrations. The number of times a rendezvous extent is scaled is called the \textit{notch} of the registry.

Take Figure 2 as an example. Suppose that on a 2-dimensional surface the \textit{jazz} tag hashes to the rendezvous extent 210 (the small highlighted extent in the leftmost subfigure). This is referred to as notch 0. All peers that register \textit{jazz} route their registrations to the RP that owns that extent. If the tag is common, the storage limit of the RP will soon be reached so it will increase the rendezvous extent to 21 by omitting the trailing 0. This extent, at notch 1, is $2^d = 4$ times the size and encompasses four peers (assuming the ICE surface is equally partitioned), each of which becomes an RP for the \textit{jazz} registry and stores a fraction of the registrations. This process can continue as needed to comfortably accommodate all registrations stored in the registry. Eventually the rendezvous extent can reach the universal extent which contains the entire surface, such that every peer in the system may store a fraction of the registrations. The benefit of this approach is that the available storage naturally scales with the number of peers registering in the system.

The scaling of a rendezvous extent does not occur simultaneously across all peers, because SPICE is a distributed system. Each peer acting as an RP waits until it reaches its storage limit before distributing registrations to the peers in the next container. Hence, some parts of the container scale ahead of others. Figure 3 shows a tract within a container scaling to the next notch.
4.2 Registry replication

While registry distribution reduces the storage load and outgoing load per cast on RPs, registry replication is designed to reduce the incoming load on peers caused by casts frequently using the same tags in their target expressions. Each peer records the frequency of casts it receives. When this exceeds the frequency limit, the peer replicates its registrations to other parts of the surface using the ICE routing algorithm. These replicas then resolve a fraction of new casts in the same way as the original RP, reducing its incoming load.

The extents to which registries are copied are found by replacing the head of the rendezvous extent address with a number of “wildcards”. For example, Figure 4 shows a 2-dimensional surface with the RP at 210 holding the registry for the jazz tag handling many casts (at left). Level 1 replication of this rendezvous extent, *10, copies the registry to corresponding extents in the other three quadrants, i.e., 010, 110 and 310. Level 2 replication makes a total of 16 copies at corresponding extents within the next deepest set of extents. **0.

Every level of replication reduces the probability of a particular replica being chosen by a factor of $2^d$, since replication replaces the first digits of an address with wildcards and is thus dependent on the surface dimensionality. A peer can calculate the replication level needed to reduce a tag’s selection frequency to a value less than its frequency limit.

Replication level $r$ for a tag with selection frequency $f$ and peer frequency limit $m$ is defined by Inequality 2. Equation 3 finds the smallest integer $r$ that satisfies this inequality, which is the level to which a peer replicates the registry.

$$\frac{f}{2^{dr}} \leq m \quad (2)$$

$$r = \left\lceil \frac{\log \frac{f}{m}}{\log 2^d} \right\rceil \quad (3)$$

When a peer determines that its frequency limit is exceeded by incoming casts, it begins the replication process. If the registry to be replicated is not distributed, the peer simply routes a copy of it to the
set of extents that will act as the roots of the replicas. Casts may be resolved by the replicas once the routed registry is received and stored by the new RPs. If the registry to be replicated is distributed, a replication command traverses the whole registry in the same way as casts. Each distributed RP that receives the command routes its fraction of the registry to the set of new replica roots. As the new replicas receive the casts, registry distribution is employed as normal to create newly distributed replicas of the registry.

Although conceptually straightforward this approach may be costly for large distributed registries. Each RP stores at most \( s \) registrations, where \( s \) is the storage limit parameter. Therefore, a distributed registry containing \( m \) registrations will be distributed over approximately \( \frac{m}{s} \) RPs. Each of these must route its registrations to the new replica extents. If \( m \) is large and \( s \) is small, this could result in thousands of routes to replicate an entire distributed registry.

This can be optimised by bundling the registrations. Instead of each RP routing its registrations directly to the new replica extents, it routes them to a nearby marshalling peer which collects many such messages before routing them together to the replicas. The marshalling peers are simply the owners of nearby potential replica extents for the tag of the registry. Note that these are not actual replicas, but extents where replicas could potentially exist at a sufficiently deep level. The level chosen determines the tradeoff between the number and size of the messages eventually routed to the replicas. That is, a high level means more marshalling peers route smaller messages. A low level generates a small number of large messages.

Each marshalling peer is responsible for bundling \( O(2^d) \) messages, for some \( j \), because of the hierarchical nature of ICE. Therefore, the total number of messages that must be routed to replicate a registry can be reduced to \( O \left( \frac{m}{s^j} \right) \), with a corresponding increase in the size of each message. The value of \( j \) must therefore be chosen to balance these two extremes. Note also that a registry need only be fully replicated once. After it has been replicated, any new registrations are routed to all replicas as they occur (or soon after).

To actually spread load to the replicas, it is necessary to adjust the cast algorithm slightly. Ordinarily, casts are routed from the publisher to the distant rendezvous extents of tags in the target expression. Replicas may be found during this route by perturbing slightly from the most direct path so as to pass through extents where a replica may exist. This is practical since it is possible to calculate the locations of potential replicas from the tag alone. If a replica is found, the route is terminated resulting in a shortened path and reduced total incoming load on the original rendezvous peer. If no replicas exist for a tag, the message will still reach the original rendezvous extent, having deviated only slightly from the optimum route.

Specifically, the cast is first routed to the nearest extent that may hold the most heavily replicated version of the registrations. If no replica is found at this extent upon arrival, it is known that the registrations have not been replicated to that level, so a lesser level of replication is selected and the process is repeated. Figure 5 shows the process. For those rendezvous extents which have been replicated, this modification
to the cast algorithm means that casts will always be
delivered to the closest possible replica (illustrated in
Figure 4). Therefore, replication combined with the
modified cast algorithm divides the amount of incom-
ing traffic for a tag evenly over all replicas, assuming
publishers are randomly spread through the network.

The altered cast algorithm may seem to promote
heavy loading of routes between potential replica
sites. In fact this is not a problem because any cast
that is sufficiently popular to create loading between
replica sites naturally triggers registry replication.
Hence replicas are found quickly on these routes, and
no worn paths appear.

4.3 Distribution and replication algo-
rithms

Table 3 and Listings 3–6 present the extended SpICE
Unloaded objects and algorithms. A new object, Registry,
is introduced which holds a set of registrations and is associated with a particular combination of tag, notch and replica_root. Registry objects
are stored by all peers that are part of a registry,
distributed or otherwise. The replica_root is the
extent that is used as the root of the registry. The
depth of this extent indicates the level of the replica-
ation. A root at the universal extent indicates a
registry is the original version, not a replica.

Peers register by constructing a Registry object
for each tag and route it to the corresponding RPs
as before (Line 10 of Listing 3). When it arrives it
is concatenated to the existing registry object (Line
8 of Listing 4). If the registry is full, it is closed
and all its registrations are routed to the next notch
(Line 15) recursively until enough space is available.
These notches are found by scaling the peers’ RPs
within the next largest container extent (illustrated
in Figure 3, and implemented at Line 21).

Registrations are routed to these tracts using a
modified Ice routing algorithm. Recall that Ice or-
dinarily routes a payload to all peers within a des-
tination tract. However it also provides two hooks
that allow an application (such as SpICE) to alter
these payloads at key points during the routing pro-
cess. Listing 5 presents the two callbacks used to
alter the registry payload as it is routed. Essentially,
these ensure that the registrations contained within
the registry payload are uniformly divided among the
peers in a destination tract. The premise is that if a
routed message is branching to a number of clusters,
the registry payload should be equally split among
them. Similarly, when a registry is to be delivered to
a peer, only a fraction of registrations proportional
to the remaining destination tract are kept. The rest
are routed to the remainder of the destination. No
registration is delivered to more than one peer, and
no registrations are omitted.

Casts are handled similarly. A peer initiates a
cast by handling it as though it were just received
from another peer (Line 19 of Listing 3). Because
the cast is not initially assigned to any particular
replica, the nearest potential replicated rendezvous
extent is calculated (Line 28 of Listing 6). This con-
tinues until a replica is found (Line 5). If it is a
non-distributed, open registry, notifications can be-
gin immediately (Line 18). If the registry is closed,
then it has been distributed to higher notches, and
the cast is routed to the scaled notch in the same way
as the registration algorithm (Line 16). Eventually,
the open registries will be reached, and consumers
will be notified.
Table 3: Extended object structure for SPICE Unloaded algorithms.

(a) A Registry object.

<table>
<thead>
<tr>
<th>Registry</th>
<th>tag</th>
<th>notch</th>
<th>replica_root</th>
<th>registrations</th>
<th>closed?()</th>
</tr>
</thead>
</table>

(b) A Registration object.

<table>
<thead>
<tr>
<th>Registration</th>
<th>peer</th>
<th>tags</th>
<th>registry</th>
</tr>
</thead>
</table>

Listing 3: The SPICE Unloaded publisher/consumer algorithms.

```ruby
1  # Peer p registers.
2  def reg(p)
3      p.registration.tags.each do |tag|
4          registry = Registry.new
5          registryregistrations.push(p.registration)
6          registry.tag = tag
7          registry.notch = 0
8          registry.replica_root = Extent.universal
9              p.registration.registry = registry
10             p.route(tag rp, p.registration)
11          end
12       end
13
14  # Peer p casts c.
15  def cast(p, c)
16      c.target.or_terms.each do |term|
17          c.term = term
18          c.tag = term.first
19              handle.cast(p, c)
20        end
21      end
```
Listing 4 The SPICE Unloaded register algorithm.

```python
# Peer p handles routed registration r.
def handle_reg(p, r):
    # Look up an existing registry at this peer.
    if (rp_registry = p.find_registries(r.tag, r.notch, r.replica_root))
        if rp_registry.closed? # The registry is closed; route up.
            route_reg_up(p, rp_registry)
        else # Add new registrations to existing registry.
            rp_registry.registrations.concat(r_registry.registrations)
    else # Couldn't find an existing registry, so create one.
        rp_registry = p.add_registries(r.registry)
    end

    # If registry is full, close it and scale up.
    if rp_registry.size > p.storage_limit and rp_registry.notch < max_notch
        route_reg_up(p, rp_registry)
        rp_registry.close
    end
end

# Peer p routes registry to the next notch for random storage.
def route_reg_up(p, registry):
    # Find the registry's current container.
    container = registry.tag rp.translate_to(registry.replica_root)

    # Scale up the part of this peer's tract in the container.
    destination = p.tract.scale_all_within(container)

    # Route the registry to that part of the scaled container.
    r = Registries.new
    r.registry = registry
    r.registry.notch += 1
    p.route(destination, r)
end
```
The callbacks for the SPICE Unloaded register algorithm.

# Deliver only a portion of the registrations to a peer.
def callback_deliver(p, route):
    if (route.destination_tract - p.tract).empty:
        # If this peer is the last in the destination tract,
        # give it the remaining registrations.
        yield route.payload
    else:
        registrations = route.payload.registry.registrations
        # Keep some registrations for this peer, proportional to
        # its size compared to the entire tract being visited.
        k = (p.tract.volume / route.destination_tract.volume) *
            route.payload.registry.registrations.size
        new_payload = route.payload
        new_payload.registry.registrations = registrations[0..k]
        yield new_payload
        # And route the remaining registrations on.
        route.payload.registry.registrations = registrations[k..-1]
    end
end

# Route only a portion of registrations to each cluster.
def callback_branch(p, route, clusters):
    # Otherwise route some of the registrations to each cluster
    # in the tract remaining to be visited.
    regs = route.payload.registry.registrations
    total_volume = clusters.inject(0) { |sum, cluster| sum + cluster.volume }
    # Send a portion of registrations to each cluster.
    clusters[0..-1].each do |cluster|
        # Make new payload with partial registrations.
        new_payload = route.payload.dup
        k = regs.size * (cluster.volume / total_volume)
        new_payload.registry.registrations = regs.slice!(0..k)
        # Route it to a cluster.
        yield new_payload, cluster
    end
    # And the remainder to the final cluster.
    final_payload = route.payload.dup
    final_payload.registry.registrations = regs
    yield final_payload, clusters.last
end
def handle_cast(p, c):
    # Test whether this peer is storing a replica of the registry.
    if c.replica_root.nil?
        root = max_replica_root(p.registries, c.tag, c.notch)
        unless root.nil?
            c.replica_root = root
            c.tract = Tract.new(c.tag, rp.move_to(c.replica_root))
        end
    end
    # A replica has been found; notify consumers or route up the notches.
    if c.replica_root
        if (reg = p.find_registry(r.tag, r.notch, r.replica_root))
            if reg.closed?
                route_cast_up(peer)
            else
                registrations = reg.find_all do |r|
                    c.target.selects?(r)
                end
                registrations.each do
                    |r| p.notify(r.peer, c) if minimum_term?(r, c)
            end
        end
    else
        # Looking for replica — find next possible replica on path.
        p.route(calc_next_replica, c)
    end
    # Peer p routes cast c to the next notch to find registrations.
    def route_cast_up(p, c)
        root = c.tag.extent.move_to(c.replica_root)
        container = root[0..c.notch]
        destination = Tract.new(c.destination, intersect(p.tract))
        destination.notch_up!(container)
        c.dup = c.dup
        c.dup.notch += 1
        c.dup.destination = destination
        p.route(destination, c.dup)
    end
4.4 Peer flux and self-stabilisation

This section examines how SPICE handles peers joining and leaving the system, whether intentionally or catastrophically. Registry distribution and replication necessarily complicate matters but the two techniques are independent so their behaviour can be considered separately.

4.4.1 Registry distribution

A new peer claiming part of a tract owned by a peer participating in a distributed registry is handled by copying all closed registries to the new peer, and splitting any open registries equally. Each peer is then capable of handling casts as normal.

Registrations can be removed by routing unregister requests to an RP, which then routes it through the distributed registry in the same ways as casts. To improve efficiency, RPs may periodically forward these casts in bundles, or piggy back them on casts, rather than route each as it arrives. Alternatively, registrations could be leased meaning they automatically expire after a period of time. Unregistration would not be required, but consumers would need to reregister periodically.

The departure or failure of a peer that is part of a distributed registry is handled as it is in basic SPICE. Peers are seeded with copies of neighbours’ data when registrations first occur, so a neighbour is able to take over both a failed peer’s tract and its role as part of the distributed registry. If both peers have open registries for the same combination of tag, notch and replica root, they are merged. If either is already closed, both are closed and any registrations are routed to the next notch contained by the remaining peer’s new total tract. This ensures that the remaining peer still has exactly one registry for each combination of tag, notch and root. In essence, local repairs to the surface and delegation of roles result in the remaining peer acting as if it had always been the only peer at that location.

General peer flux and the removal of registrations by consumers may eventually lead to distributed registries becoming fragmented and only partially full. It is inefficient for casts to be routed to all RPs whenever a lesser number would suffice. For this reason, peers can initiate a redistribution process if the number of registrations they store falls below a threshold. A peer does this by routing a redistribution command to those RPs above it which propagates to the registry’s leaves in the same way as a cast. Upon receipt of this command, each RP routes its registrations to the original RP where they are treated as new registrations and redistributed as normal. During this period, old RPs maintain copies of the registrations so that casts can continue to be resolved before they have been redistributed. After a short period of time, the registrations are presumed to have been stored and the old RPs delete their registries.

This algorithm potentially permits consumers to receive duplicate casts or miss casts completely, thereby failing to satisfy both the safety and liveness conditions of IGM. The liveness condition can be satisfied by using a distributed commit protocol between the old RPs and the original. Only once the original has agreed to take responsibility for consumer notifications does the old RP delete its registry. Note that although consumers will not miss any casts, this still permits some to be notified twice if a cast occurs during this commit. Such duplicates can be detected by the SPICE code running on the consumer and eliminated to ensure applications do not receive them.

4.4.2 Registry replication

Registry replication can be applied to both distributed and non-distributed registries. Each replica behaves exactly like the original registry, distributing as needed to efficiently store incoming registrations. The algorithms for maintaining distributed registries described above apply equally to all replicas.

A new peer may register at any replica. Because the IGM liveness condition only requires that registrations are eventually (not immediately) active, replicas need only be weakly consistent. I.e., registrations can be collected for a period before they are routed to the other replicas. Similarly, requests to unregister can be collected and copied to other replicas periodically.

When a replica’s incoming cast frequency drops below a threshold, it tries to reduce the level of repli-
cation in the system to avoid the maintenance traffic required to synchronise registries. It does this by routing a suggestion to all other replicas that the replication level be reduced. If no replica vetoes the suggestion within a certain period, each replica reduces its level. In most cases this means the registries stored by the replica can be deleted, although replicas situated at the rendezvous extents for the reduced replica level continue to resolve casts.

5 Conclusion

This report has described SPICE, an IGM implementation in which every participant potentially serves IGM casts to consumers. SPICE is founded on a distributed, structured overlay network called ICE which combines an hierarchical tesseral addressing scheme with an amortised routing algorithm. SPICE employs many rendezvous points on the ICE surface to store tag registrations and marshal casts for delivery to consumers.

Advanced distribution and replication techniques are designed to limit the amount of work performed by participating peers. Registry distribution reduces storage and outgoing load on peers by storing registrations at peers near to the original rendezvous points, exploiting ICE’s hierarchical tesseral addressing. Registry replication reduces incoming load and total outgoing load on rendezvous peers by making complete copies of registries at corresponding extents on the ICE surface. These replicas can be discovered by casts in the course of routing towards a rendezvous extent.

Acknowledgements

The authors would like to acknowledge the support of the Smart Internet Technology CRC.

References


