Computer Science and Information Systems 11(2):549–567 DOI: 10.2298/CSIS130823024S

Iris: A Decentralized Approach to Backend Messaging Middlewares

Péter Szilágyi[1,2]

[1] Eötvös Loránd University
1053 Budapest, Hungary
[2] Babeş-Bolyai University
400084 Cluj-Napoca, Romania
peterke@gmail.com

Abstract. In this work we introduce the design and internal workings of the Iris decentralized messaging framework. Iris takes a midway approach between the two prevalent messaging middleware models: the centralized one represented by the AMQP family and the socket queuing one represented by ZeroMQ; by turning towards peer-to-peer overlays as the internal transport for message distribution and delivery. A novel concept is introduced, whereby a distributed service is composed not of individual application instances, but rather clusters of instances responsible for the same sub-service. Supporting this new model, a collection of higher level messaging patterns have been identified and successfully implemented: broadcast, request/reply, publish/subscribe and tunnel. This conceptual model and supporting primitives allow a much simpler way to specify, design and implement distributed, cloud based services. Furthermore, the proposed system achieves a significant switching speed, which – given its decentralized nature – can be scaled better than existing messaging frameworks, whilst incurring zero configuration costs.

Keywords: peer-to-peer, decentralized, message oriented middleware.

1. Introduction

A message-oriented middleware (MOM) is either a hardware or software infrastructure component, with the sole purpose of removing the complexity of communication from a distributed system, allowing individual network components to focus on their specific tasks [2].

The core concept behind MOMs is based on the observation that classical, stream-based communication introduces a very tight coupling between network components, requiring significant efforts to develop and maintain distributed systems. Instead, in MOM based systems the unit of communication is a message, *an undividable, self-contained block of data propagating from sender to recipient(s)*. As the transferred data is self contained, the network can evolve dynamically without costly protocol setups and teardowns.

The second core concept is the middleware part, whereby all distributed clients are in contact only with the MOM, but not each other. The most important implication is that the middleware is responsible for transferring application messages from origin to destination, whilst clients remain ignorant of the routing complexity. This decoupling grants the whole system both simplicity and greater flexibility in terms of scalability and environmental heterogeneity [8].

The last (optional) concept in MOMs are message queues, which provide further decoupling by allowing applications to communicate asynchronously [23], not requiring the communicating peer to be accessible at the time of messaging. However, for distributed systems targeted by this paper (i.e. dynamically scaling back-end services), persistent messaging is not a requirement.

Due to the significant popularity of cloud computing [3], messaging middlewares gained an even bigger role in distributed system infrastructures. Whilst previously internal back-end systems could have used arbitrary network topologies, with the advent of clouds, system designers are forced to think in more general – and many times, less reliable – solutions. Messaging middlewares provide the extra simplification of distributed systems to keep focus on the problem and prevent it from shifting towards cloud communication.

Even though the operational environment of cloud providers, as well as operational requirements of cloud services are drastically different from *pre-cloud* ones, the same underlying messaging models are continued to be used. We argue, that although these models are perfectly feasible, not taking advantage of cloud specifics leads to suboptimal distributed systems. Essentially, cloud platforms encourage the *load-based elastic evolution of hosted services*, where deployed applications must be ready to scale at a short notice. This is the weakness of current messaging models, as they were not designed with constant scaling in mind. To prepare for such scenarios, all non-scaling concepts need to be eliminated: location and cardinality.

This paper presents a MOM model, design and algorithms that significantly simplify the development of distributed back-end services. Firstly, a novel networking abstraction is introduced, whereby the smallest unit of composition in a distributed system is a *cluster of instances*, opposed to individual instances in previous literature. Secondly, four core communication patterns are defined, which are essential for back-end services and are also fully compatible with the *clustered* compositional model: request/reply, broadcast, tunnel and publish/subscribe. Finally, the feasibility of the model and messaging schemes are demonstrated through peer-to-peer networks, fully developing a P2P overlay supporting the required operations, and at the same time requiring zero configuration and maintenance. The P2P related challenges of decentralized bootstrapping and peer-to-peer security have already been covered in two previous papers [21,22], the present one focusing solely on decentralized routing.

The paper starts out by presenting the existing solutions, continuing with the core communication patterns and reliability guarantees identified as essential for back-end services. Afterwards, the implementing models, algorithms and optimizations are presented. Finally, the proposed solution is validated through a series of benchmarks, confirming the model's feasibility.

2. State of the art

Historically, hard requirements were placed on these messaging middlewares: the guarantee of data security and integrity in any operational environment; the guarantee that no messages are lost in the face of any network, software or hardware failure; yet to still achieve a significant switching throughput.

The most prevalent technology to have met these criteria is the Advanced Message Queuing Protocol (AMQP) [24,14], with RabbitMQ[3] being one of the leading implementations of the specs. However, the reliability guarantees AMQP undertook blew up the protocol complexity enormously, leading to centralized solutions that are hard to scale [10].

iMatix took a new approach to messaging – diverging from AMQP – by removing the concept of message brokers and instead, placed the message queues directly into the client processes with their ZeroMQ library[4], coining the term *"sockets on steroids"*, arguing that all messaging should happen at the endpoints [11,19]. The main issue with iMatix's approach is that they reintroduced the tight coupling that MOMs originally set out to remove, and although ZMQ provides higher level communication primitives, these are bound to socket level, forcing the user of the library to define and implement the needed network topology. This works well in static environments, but with the prevalence of clouds (massive distributions and massive failures), the administrative overhead of a custom, user-managed topology becomes a significant cost [18].

A different initiative is the Data Distribution Service for Real-Time Systems (DDS) a standardized MOM specification [15,17], with RTI Connext DDS[5] being one of the most advanced implementation of it [12]. As its name suggests, the primary goal of DDS is modeling complex data flows through which data can be disseminated to interested parties in a large network of nodes. This is achieved using a publish/subscribe model, where the physicality of the network is hidden from participating nodes and all addressing is done through so called topics. Although very powerful, the publish/subscribe abstraction models only information flows, but does not cater for other messaging patters – such as request/reply or load balancing – which are essential for implementing back-end services.

A final MOM needed mentioning is the eXtensible Messaging and Presence Protocol (XMPP), an IETF standardized protocol[6] focusing on data exchanges relating to instant messaging, presence management and social collaboration. It can be considered a generalized routing protocol for XML data. However, its goals are very different from the back-end service ones targeted here.

3. Proposed abstractions

To retain as much flexibility as possible, our messaging model assumes only a bare-bone cloud environment. Such a cloud service-model is commonly called *Infrastructure as a Service* (IaaS), and provides the consumer with the capability to provision fundamental computing resources (e.g. processing, storage, network, etc.) and run arbitrary software on them [13].

3.1. Distribution model and unit of composition

Originally, back-end services used vertical distribution: the logically distinguishable parts of a complex service were split into individual component applications, each hosted either together, or separately based on their resource consumption.

[3] `http://www.rabbitmq.com/`
[4] `http://zeromq.org/`
[5] `http://www.rti.com/products/dds/index.html`
[6] `http://xmpp.org/xmpp-protocols/rfcs/`

This solution has however quickly shown its weaknesses to failures, as well as its limitations under high load. System designers hence extended the vertical model with the concept of horizontal distribution: multiple instances of the same service components are run simultaneously on different machines with tasks distributed between them according to some load metric.

Whilst theoretically sound, the extended model incurs significant administrative costs, since each entity must be able to contact its own servicing components and load-balance between them. The addition or removal of instances further complicates overall component logic and maintenance.

We introduce the high level abstractions required to retain the simplicity of the vertical distribution, whilst obtaining the flexibility of the horizontal distribution. This is done by turning towards peer-to-peer networks as the base communication model (Fig. 1).

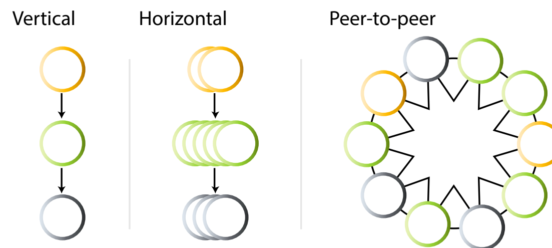

Fig. 1. The vertical model is the theoretical target, whilst the horizontal one is the operational goal. The paper presents a solution based on the last model, achieving the best of both worlds.

State of the art distribution models consider an application *instance* (e.g. web-server, database, etc) the smallest unit. We argue that this is the most significant design flaw responsible for the increased complexity of distributed systems: distributed communication is too complex to keep track of individual instances.

We propose a novel abstraction, where the *cluster* is at the lowest level: a *group of applications responsible for the same sub-task* of the service (e.g. collection of databases). At any point in time there can be any number of instances belonging to the same cluster (including zero); it is all the same from the consumer's point of view: a reply either arrives or it does not.

These clusters can then be assembled into a *service*, where any component can query another cluster for some sub-service, without having to know neither where, nor how many instances can handle its requests. Such a *service* is considered the unit of security [22].

Finally, multiple such services can be composed into a *federation* or a *cloud*, crossing security boundaries and allowing inter-service communication (shortly expanded in the future works section).

3.2. Communication patterns

In order to support the proposed distribution and composition model, four communication patters have been identified: *request/reply*, *broadcast*, *tunnel* and *publish/subscribe* (Fig. 2). These are refinements over author's previously identified ones in [20].

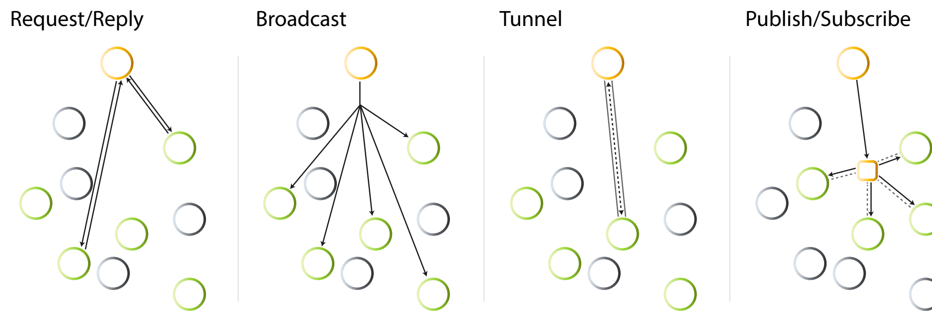

Fig. 2. Core messaging patterns to support the model. The first three accomplish standard back-end communication, whilst the last one enables more specialized operations.

The first pattern is the *request/reply*, a natural communication pattern for the target use-case of interconnected services: a consumer sends a request to a sub-service, which responds with a reply. The proposed model adds a twist, which is a direct consequence of the *cluster* being the smallest unit of composition: whenever a request is made, the target is not a single entity, but rather the cluster of entities serving the same purpose. It is the responsibility of a messaging middleware implementing this model to make sure that an appropriate node gets the request, taking load balancing into consideration too. The consumer should be aware of neither the serving cluster's size, nor individual component's whereabouts, but the name/type of the cluster alone (e.g. *'databases'*) must be enough to process the request.

The second pattern is the *broadcast*, a supporting scheme besides the *request/reply*. The difference between the two is, that whilst a request is delivered to a single member of the cluster, a broadcast is forwarded to all of the participants. This is the only way to contact every instance within a group. But since there is no concept of member count, there is no possibility for individual responses, as the recipient would never know how many replies to wait for. Hence broadcasting is an asynchronous, one-way operation.

The third pattern is the *tunnel*, which itself is another supporting scheme of the cluster communication. Its goal is to solve the challenge of stream communication and/or stateful protocols, where an operation consists of multiple data exchanges (e.g. a database transaction). A *tunnel* establishes a communication stream between a client and a member of a cluster, with ordered and throttled message delivery, persisting until either side closes the connection.

The messaging patters enumerated above solve most of the communication requirements of a back-end service. The last scheme, the *publish/subscribe*, is a very specialized one, as it allows breaking cluster boundaries. The underlying concept is well known in the literature [5]. Any instance within the network may subscribe to a *topic* (any number of

them), effectively forming temporary clusters. Any node in the network can then publish events to these clusters the same way as *broadcast* does.

3.3. Reliability guarantees

The reliability guarantees of a distributed messaging model are *the assurances it can always satisfy about message delivery in the presence of software, system or network failures*. From this point of view, the proposed model takes a significantly more modest standpoint compared to previous messaging models (e.g. AMQP family). Most of such requirements cannot be handled at the messaging level, and trials usually result in complexity explosions [10]. This paper instead focuses on the robustness of the messaging layer, providing only those guarantees that are essential and natural to the respective level of abstraction.

The *request/reply* pattern has two possible points of failure. The first scenario of them is that the request is lost during transit or that there are no available instances to respond. This is impossible to circumvent in a loosely coupled environment, since any hardware failure will result in message loss. The second scenario is when the request is successfully serviced, but the reply is the one lost. The reason this problem is impossible to solve at the messaging layer is twofold. Per the Two Generals' Paradox [1], both of the endpoints of a request/reply exchange can never be sure of the operation's success, hence a reply cannot be cached for resending. Secondly, without the guarantee of idempotence, neither can the request be cached for automatic resending. This means, that transit failures can be detected at the messaging layer (through timeout mechanisms for example), but their handling must be delegated up to the application layer.

The *broadcast* and *publish/subscribe* schemes are sensitive to network partitioning, where some of the addressed instances receive a message, but not all of them (possibly even none). Although unfortunate, this issue persists in any distributed environment, and can be solved solely using a central tracker. We argue, that reliance on such guarantees is a fundamental design flaw in most distributed systems, and as such, the proposed communication model makes no attempt to provide it at the messaging layer. Of course, a high-level client application can easily implement such behavior on top of the core schemes with the exact guarantees needed (bearing the necessary sacrifices).

Finally, opposed to the previous patterns, the *tunnel* is a high reliability messaging primitive. It provides the very same guarantees that a transport layer TCP connection does: as long as the connection is alive, the messages are guaranteed to be delivered eventually and in the same order as sent. Besides ordering, the data flow is also throttled to prevent overloading the recipient.

4. Overlay network

The Iris overlay adopts a fully decentralized peer-to-peer model, where – given the one messaging node per host constraint – each participant has equal responsibilities. To achieve the desired messaging patterns, it builds application *clusters* based on the multi-cast trees formed by Scribe [7], which in turn are based on the routing paths of Pastry [16].

4.1. Foundations: Pastry and Scribe

Pastry is a generic, efficient, scalable, fault resilient, and self-organizing peer-to-peer substrate [6], forming a highly robust distributed hash table (DHT). The goal of the Pastry DHT is that, given a message and an associated routing key, to deliver the message to that specific network participant, whose node identifier is numerically closest to the key itself.

To achieve this, each participating node is assigned a unique, uniform random node id, taken from a circular, 128-bit identifier space. Each node is then considered responsible for the slice of the id space circularly the closest to it. Furthermore, each node maintains a local routing state: Leaf-set (numerically closest peers), Routing table (peers with matching prefixes of $1..N$ digits) and Neighborhood (physically closest peers).

Given the above routing state information, at each routing step a node checks the common prefix (length l) of the message's routing key with its own node id, and then selects a node from its routing table which shares at least $l + 1$ digits with the message. If such a node is not present within the routing table, the leaf-set is consulted and the numerically closest peer receives the message. If according to the leaf-set, the current node is the closest, the message is delivered upstream for higher level processing (Fig. 3).

Of course, Pastry achieves self-organization by dynamically maintaining the routing state within each node, these synchronizing between each other whenever a churn event is detected (a node joins or leaves). For such fine grained details about the protocol we refer the reader to the original paper [16].

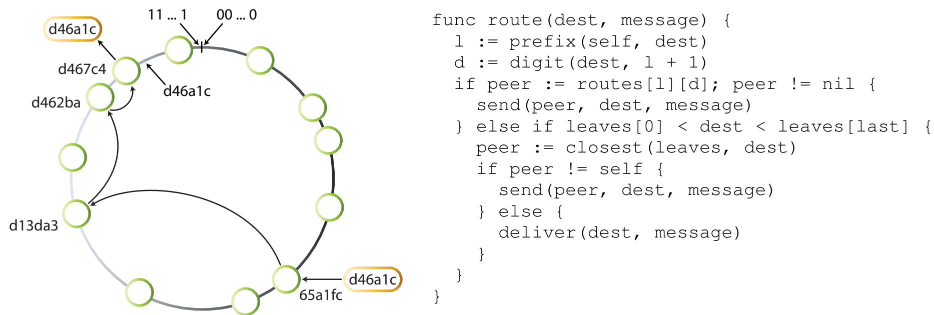

Fig. 3. Pastry routing algorithm and example. A message with the routing key *d46a1c* is forwarded from node *65a1fc* towards its destination. The message is prefix routed, decreasing the distance from its destination exponentially in each step. At the very last step no longer matching prefix is found, so the leaf-set is consulted and the message delivered to its final destination.

Pastry was originally conceived for internet-scale networks. However, in our proposed messaging scenario we assume that a service is running in its entirety within one data center[7], due to which two modifications were made. Firstly, the concepts of proximity and associated neighborhood set were removed, since within a single cloud the latency/bandwidth differences between links are not significant enough to warrant the complexity. Secondly, with the reduced node count of back-end services – in the range of hundreds – an ID space of 128-bits leaves the routing tables mostly empty. It should thus

[7] It will be the responsibility of a *federation* to span data centers.

be reduced as much as possible, while keeping the probability of a random ID collision insignificant. According to the birthday paradox [9], a 40-bit ID space S for 10K nodes N results in a collision probability $p(N, S)$ of 4.5483×10^{-5}.

$$p(N, S) = 1 - e^{-N^2/(2S)}$$

Scribe is a decentralized, scalable application-level multicast infrastructure, built on top of the Pastry overlay [7]. The goal of Scribe is to allow nodes to create *groups*, which can then be joined/subscribed by any peer within the network. Messages sent to these groups are delivered to all members.

To achieve this, whenever a node wishes to create a new group, the name of the group is mapped to a Pastry identifier, after which Scribe uses Pastry to route a *create* message to the node responsible for that specific ID, which implicitly assigns the group to the specific node. This node will assume responsibility for it, becoming the group's *rendez-vous point* (Fig. 4a).

After group creation, any participant of the Scribe network can become a member of the group by routing a *join* message towards the group's *rendez-vous point*. Opposed to the *create* message, the *join* is not forwarded blindly between nodes, but instead each node along the path to the group owner maintains a local subscription list, and if the current node is already part of the group, then the new node will be attached to it. If the traversed node is not a member of the group, it will create a new local subscription, terminate the arrived join, and initiate a new join itself. This procedure will create a tree structure for every group, rooted at the *rendez-vous point* (Fig. 4b).

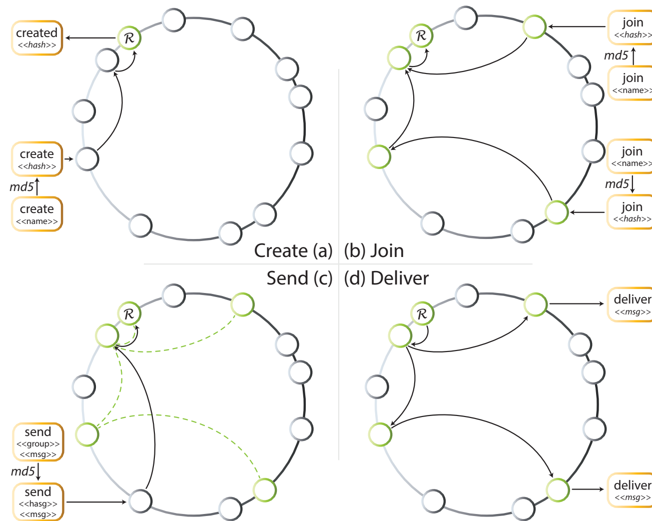

Fig. 4. Scribe group operations.

Given the freshly built subscription tree, whenever a message is to be disseminated to the group members, it is first routed to the group's *rendez-vous point*, which will initiate

a distribution along the edges of the tree, from parent to child, each internal node of the tree in its turn also forwarding the message to its children (Fig. 4c & 4d).

Since Scribe was conceived for public peer-to-peer networks, it contains an underlying credential system to limit group membership and communication to authorized entities only. The proposed system uses a different authentication mechanism [22], whereby a successful connection already pertains the granting of full privileges. Removing the credential sub-system makes group creation redundant, since group joining will already entail all the necessary steps. Due to the same reason, during content distribution, the messages do not necessarily have to reach the root node before dissemination can begin: any subscriber receiving such a message can immediately begin distributing to both children and parent, saving valuable network hops.

4.2. Iris overlay

Iris is a decentralized, application-scope, group communication platform, built on top of modified Scribe and Pastry protocols. The goal of Iris is to implement a batch of high level messaging primitives to simplify the distributed communication among nodes in a back-end service.

Cluster formation and teardown As defined in section 3.1, the *cluster* is the smallest unit of composition in the Iris platform. Whenever an application wishes to communicate through Iris, it must first become a member of a cluster. Hence the first step in the life of an Iris application is joining – possibly by creating – such a cluster.

Cluster formation and membership management are based on the Scribe multi-cast groups. Whenever an application requests to join a cluster with a given name, Iris maps that cluster name to a Scribe group name by prepending a tag to it. The goal of the tag is to differentiate between other types of Scribe groups. After tagging the name, a Scribe *join* is initiated, turning the node a member of the requested group, implicitly creating the group if non-existent.

When the application finishes using the Iris network, it issues a close request, leaving the joined cluster. After the same cluster to group mapping, a Scribe *leave* is initiated, removing the application from the group subscription tree. If a node's local subscriptions all terminate, the node itself leaves the group, cascading until either a live member remains or the whole group is torn down.

Messaging primitives The simplest communication pattern is the cluster *broadcast*, specifically because Scribe already provides it out of the box. When an application broadcasts a message to a cluster, Iris first resolves the Scribe group name of the cluster, after which it delegates message distribution to the Scribe protocol according to its multi-cast subscription tree (Fig. 4c & 4d).

Request/replies are a little more tricky. The core concept is the same as with *broadcast*: Iris resolves the Scribe group name and issues a Scribe delivery. But opposed to the previous scenario, the message must be delivered to a single recipient. To achieve this, the message is routed towards the group rendez-vous point, but after entering the multicast tree it is not simply distributed to all sub-trees, but rather at every group member, a decision is made. Each member maintains the load balancing state of itself and all local

subtrees. An outstanding request is either delivered to a locally available cluster member, or routed on a single path to a subtree (details at the end of the section). When the request is finally delivered to a node and processed, the reply is routed back to the sender using the Pastry DHT directly.

The *tunnel* primitive is assembled through a combination of the previous *request/reply* pattern and direct usage of the Pastry network. When a tunneling request is made to a member of a cluster, that request is delivered the same way as a normal request. However, upon arrival, the remote side replies with its own Pastry address. At this point both parties have obtained each other's Pastry identifier and can communicate directly through the DHT. The tunnel is assembled using the same concepts as a TCP stream: each side is assigned a tunnel id (incremental throughout the lifetime of the node) and each packet assigned a sequence id. With these two information, any packet can be reliably delivered to the correct tunnel endpoint in the correct order. To prevent overloading a peer, each tunnel has a throttling mechanism, whereby each packet requires an acknowledgment from the remote side, allowing only a limited number of un-acked packages to be sent.

The last pattern, the *publish/subscribe*, is based on the same exact principle as the *broadcast*. The difference is that topic names are mapped to a separate set of Scribe groups than clusters by using a different prepended tag. This ensures that cluster and topic names will never collide with each other. Additionally, opposed to clusters, applications have to manually subscribe to topics, but can be subscribed to an unlimited number of then.

Split clustering Although the Scribe multi-cast trees supporting the Iris clusters are decentralized, since all inbound messages target the root of the tree for distribution, this root node can become a significant bottleneck.

Iris circumvents this issue by introducing *split clustering*. Instead of relying on a single Scribe multi-cast tree, every cluster and topic internally uses a number of Scribe groups simultaneously (simply adding a second tag to the name). Whenever a message enters the system, a sub-cluster is chosen using round-robin for delivery, ensuring that consecutive messages take different paths (Fig. 5).

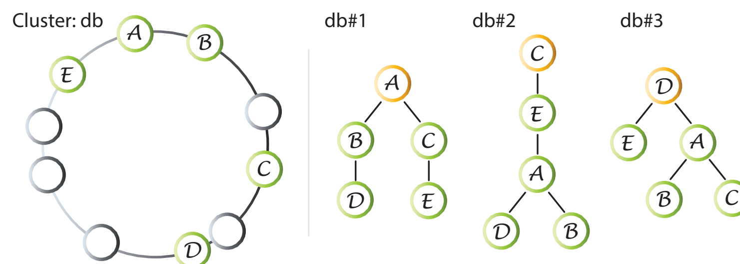

Fig. 5. A single Iris cluster split into multiple Scribe groups. On the left, the Pastry DHT can be seen with the green circles being part of an Iris group. On the right, these same few nodes can form a number of different Scribe multi-cast trees, each time with a different group rendez-vous point. Note, additional intermediary nodes may be part of these trees, but were not displayed to prevent clutter.

Load balancing As mentioned in the previous section, using the *request/reply* scheme, the framework is expected to load balance messages between all possible destinations that could handle them. In order to see how this might be accomplished, it is important to observe the internal state of the system, more specifically, that of an Iris cluster.

An Iris cluster – implemented by a Scribe group – is a rooted multi-cast tree, where each node within the tree has knowledge only about its direct children and parent. This conceptual separation, that some neighboring nodes are children and one is the parent, is essential for maintaining the tree structure, but irrelevant otherwise. Thus, from a load balancing standpoint, each node can be considered the root of a tree, where each child is a subtree of the original cluster (Fig. 6). This abstraction permits a uniform balancing algorithm for all nodes, without needing specialized logic for different parts of a tree.

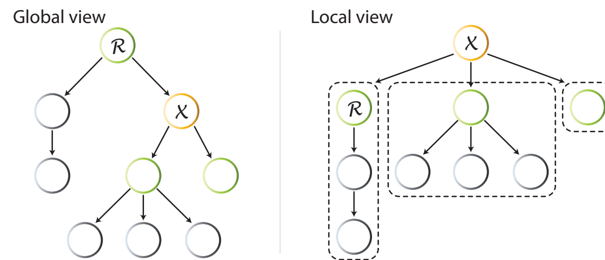

Fig. 6. Different views of the cluster. The global view is how the multi-cast tree really looks like in the network, but the local view is how an internal node sees the tree from a balancing standpoint.

With the above abstraction, the load balancing mechanism boils down to a greedy algorithm, where each node, upon receiving a request, decides whether to deliver it to a locally attached client application, or route it down to one of its local subtrees (except the one from which the message originates).

The logic which decides on the exact subtree to forward a request to is based on the *dynamic weighed round-robin* algorithm. The core concept of the round-robin algorithm is to issue every request to a different entity; but since these entities – subtrees in the current scenario – are wildly heterogeneous, specific weights are assigned to each, ensuring that the number of requests are proportional to the total available computing capacity of the subtree. Additionally, the capacities of each subtree can vary over time due to new nodes arriving, old ones leaving, or simply by experiencing an unrelated load on certain members. To cater for these dynamic scenarios, the nodes periodically exchange capacity information between neighbors, ensuring that each node has a local estimate of the capabilities of each of its neighboring subtrees.

Although this mechanism is very robust, it suffers from two weaknesses. Firstly, requests are passed on each link in both directions due to the greedy decision making. But each request matched by another – flowing in the opposite direction – could have been avoided in the first place. A theoretical solution would be a predictive algorithm, but in the presence of sporadic bursts, such an algorithm causes massive congestions. Furthermore, the effect of this weakness is only visible if the transfer of the data is more expensive that it's processing.

The other weakness is in the capacity estimation, which at the moment is based on the number of requests processed at the present load since the last periodic capacity exchange. If the time required to process a request comes close to or goes over this cycle, the load balancer will become random. Still, given the use case of back-end services, this should occur rarely and only during serious system overloads.

5. Preliminary evaluation

To analyze the performance of the suggested Iris overlay, a prototype[8] was implemented in the Go programming language and benchmarks executed in two different environments:

- For computational performance measurements, a single machine with dual Intel Xeon E5-2687W processors and 64GB of memory was used, with all messaging passing running through localhost.
- For scalability measurement, a small physical cluster of 12 machines, each containing two dual code AMD Opteron 275 processors and 4GB system memory was used, with all messaging passing through a 1 Gigabit network.

5.1. Request/reply performance and scalability

The computational broadcast benchmarks were run by starting 4 Iris nodes on the Xeon machine, with one client application attached to each. The clients formed two clusters, two in each. Furthermore, each client was a simple echo service (i.e. responds with the request itself).

To measure the performance, each cluster started issuing requests of a given size to the other, loading the system until saturation. At that point the load was kept up for 30 seconds and the throughput measured. Afterwards the system stepped to the next message size and repeated the procedure. The results were plotted on Fig. 7.

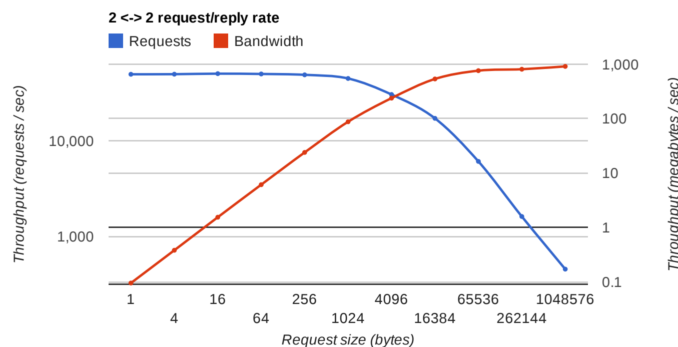

Fig. 7. Request/reply performance evaluation. Blue designates the requests served by the whole system while the red the payload bandwidth.

[8] Publicly available at `http://iris.karalabe.com`

As seen from the above chart, the system reaches a remarkable switching capacity, capping at around 50 thousand requests per second (100 thousand messages counting the replies). Looking at the bandwidth, the useful data throughput caps at over 900MB/s. It is important to emphasize that this performance is achieved with 128-bit AES encryption included between nodes [22].

For the scalability benchmark the same echo service model was used, but each machine ran a single client application and one Iris node. This benchmark was ran for systems of gradually increasing sizes from 2 to 12 hosts, and the same metrics collected.

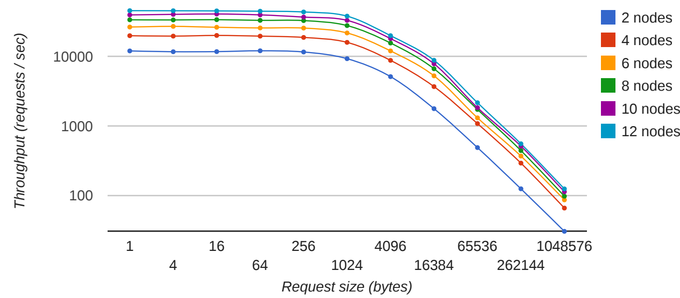

Fig. 8. Globally served requests.

The scalability results look very promising, with additional nodes linearly increasing the overall system throughput (Fig. 8): two nodes have capped at around 12 thousand requests per second and each additional pair increased this by 6K, reaching 46 thousand requests (92 thousand messages) for the full 12 nodes.

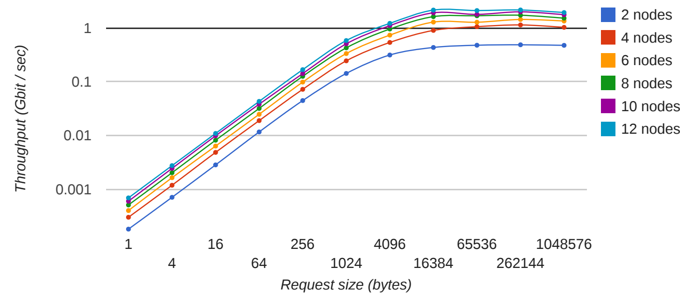

Fig. 9. Globally useful bandwidth (request + reply).

Looking at the data-rate chart (Fig. 9), the most interesting part is the bandwidth cap, which for 12 nodes is at around 2.1Gbits/sec. Since each node had a 1 Gigabit network connection, the theoretical maximum throughput is 6Gbits/sec. This means that – ignoring system messages and headers – each data packet traversed an average 2.85 network hops.

This hop count is not caused by the overlay network, but rather the inferior load balancer currently implemented.

5.2. Broadcast performance and scalability

To benchmark the broadcast operation, the same environmental setup was used as in the request tests: a number of clients form the Iris network, half joining one cluster and the other half another. The clients then started broadcasting all of the members of the opposite group.

As previously, the performance benchmarks were ran on the Xeon machine and the scalability benchmarks on the AMD cluster, plotting the delivered message and payload data-rates on Fig. 10 and Figs. 11, 12 respectively.

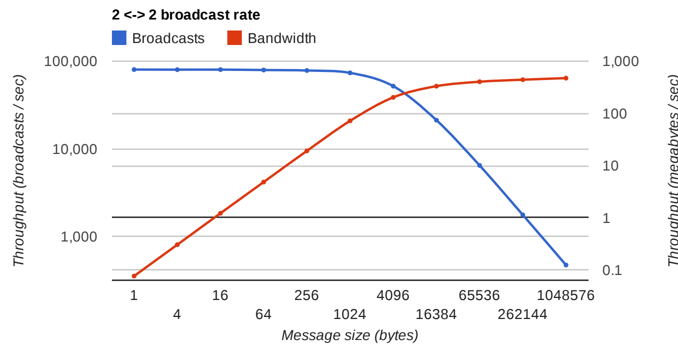

Fig. 10. Broadcast performance evaluation. Blue designates the delivered broadcasts by the whole system while the red the payload bandwidth. Note, these are twice as many as initiated since each has two recipients.

An important observation is a regression in both maximal request throughput as well as bandwidth cap. The system was able to deliver 80 thousand messages only (20% lower than using requests), and has also peaked at approx. 475MB/s throughput (down from 900MB/s).

We consider two explanations probable: since the network is very small – consisting of only 4 nodes – requests almost always get delivered to the right place, but broadcasts need to traverse the Scribe multi-cast tree. One additional hop per broadcast would be enough to halve the maximal data-rate. Another possible explanations is within the implementation details of the broadcast, namely that the handling of broadcast messages has higher memory costs both size and operation wise. Since the single machine is saturated already, extra memory allocations and copies could have an adverse effect on performance.

Looking at the scalability charts (Fig. 11), the most pleasant thing to notice, is that the total message switching capacity of the system is the same as in the request/reply benchmarks. This leads to the presumption that the above noticed regression may be a benchmark anomaly and not a real issue. Still, further experiments are needed to confirm one or the other.

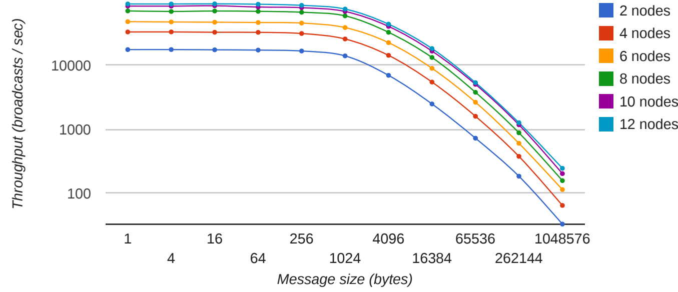

Fig. 11. Globally delivered broadcasts.

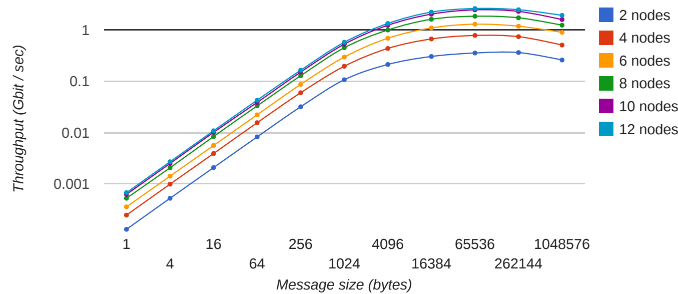

Fig. 12. Globally useful bandwidth.

On the data-rate side (Fig. 12) again we have positive results: not only did the anomaly noticed earlier disappear, but the system reached an overall data throughput of over 2.6 Gbits/s, reducing the average hop count to 2.3 per message.

At closer look, there is a small inflection on the data-rate chart when broadcasting large messages. This is deemed to be caused by a limitation of the benchmark and not the system itself and is only seen due to a combination of the AMD machines reaching their processing limits and the network being saturated simultaneously. The net effect is pulsating broadcasts, alternating between overload and starvation.

5.3. Tunnel performance and scalability

The tunnel benchmarks use the same environmental setup: a number of clients form the Iris network, out of which half join one cluster and the other half the other. Each node will then establish a tunnel into the opposite cluster and will stream messages.

It must be noted however, that this benchmark is not accurate. The problem is that these tunnels will not be distributed evenly between the nodes forming the two clusters, and because of that, overloaded nodes will produce significant delays. This could be avoided with more sophisticated benchmarks, but that would require a production quality overlay implementation.

As previously, the performance benchmarks were executed on the Xeon machine (Fig. 13). Skipping the wobbly effect cause by the aforementioned anomaly, we can observe a staggering throughput regression compared to previous messaging patterns. The highest rate achieved was 36–40 thousand messages per second, half of previous speeds.

The cause however lies within the implementation of the tunnel: to achieve ordered and throttled delivery, all messages need to be acked, which in effect matches each data packet with a system packet. The result is, that for 40K data message, the system passes 80K messages in total. This means, that for small messages, a tunnel is limited by the system noise.

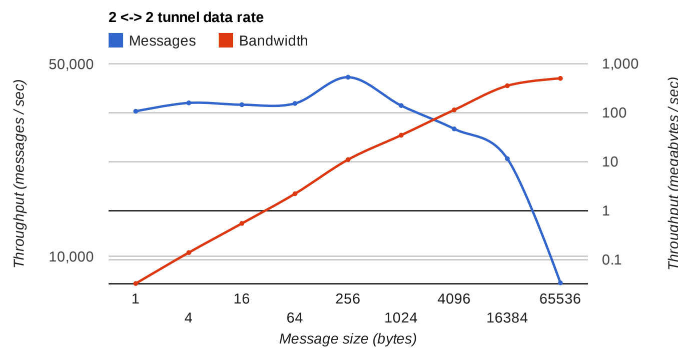

Fig. 13. Tunnel performance evaluation. The blue line is the total number of messages transferred through tunnels whilst the red is the useful payload bandwidth.

Looking at larger messages, we can see this issue alleviated a bit, with only 30% performance loss compared to the asynchronous message patterns, capping at around 500MB/s compared to the requests' 762MB/s.

The message throughput plot on the scalability charts (Fig. 14) presents both good and bad news. The good is, that – as expected – the tunnel primitive too scales linearly with the number of added nodes. The bad however, that the benchmark anomaly is getting worse with each added node.

Plotting the scalability data throughput (Fig. 15) tells a similar story to the performance benchmarks, that tunnels suffer a 36% performance loss compared to the *requests*, with the total useful bandwidth reaching 1.6Gbits/sec for 12 nodes compared to 2.2Gbits/sec.

The moral of the tunnel benchmarks is that the current *ack*-ing implementation has a serious hit on total system performance. The proposed exploration points are prioritized system messages, the splitting of larger messages into smaller manageable chunks to prevent clogging up network links and even a completely new tunnel implementation based on Iris requests and lower level direct TCP links.

The final scheme, the *publish/subscribe* has the exact same implementation internally as the broadcasts (from the message passing point of view) and hence are expected to perform identically. Due to space limitations, these have not been separately plotted and analyzed.

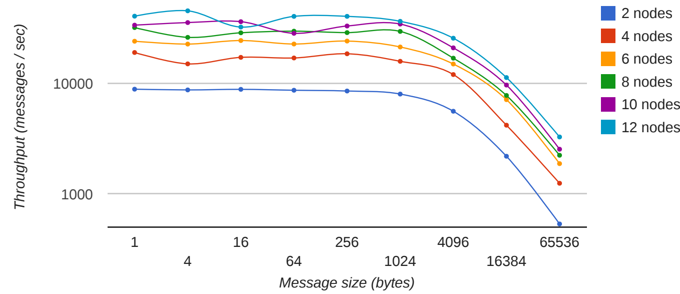

Fig. 14. Globally transferred messages.

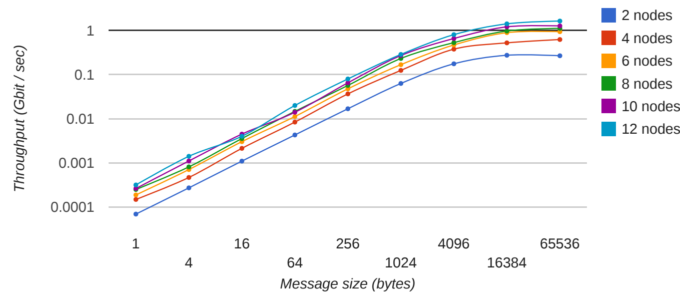

Fig. 15. Globally useful bandwidth.

6. Conclusions

This paper presented the design of the Iris decentralized group communication infrastructure based on the Scribe multi-cast protocol and the Pastry distributed hash table. The proposed overlay introduced a novel concept where *clusters* of nodes are considered the unit of composition, supporting four communication primitives: request/reply, broadcast, tunnel and publish/subscribe. These, combined with the framework's zero configuration nature, result in a significantly simplified approach to implementing decentralized back-end services.

To support the design, a preliminary evaluation was executed using a prototype implementation. Through the obtained results we conclude, that the Iris decentralization model looks very promising, reaching a simplicity yet scalability beyond previous messaging attempts. A further case study has also been carried out, assembling a decentralized ray-tracer using the Iris system. Due to space limitations, this experiment can be found as supplementary material on the Iris project's website[9].

The model presented in the paper supports a single decentralized, back-end service. It is important to emphasize that these distributed systems can be composed to create an even larger ecosystem of intercommunicating services. Depending on the desired goal, these

[9] `http://iris.karalabe.com/papers`

ecosystems can be achieved using two models. Tightly coupled services could be federated across data centers for availability and location considerations using direct links between services and having gateway nodes relay messages between them. Or more loosely coupled ones could be assembled into a cloud of services using a super-peer architecture [4], creating a Platform-as-a-Service middleware. Each of these models, however, poses significant new challenges and thus are the next steps in the Iris research project.

Acknowledgments. The research was partially carried out as part of the EITKIC_12-1-2012-0001 project, which is supported by the Hungarian Government, managed by the National Development Agency, financed by the Research and Technology Innovation Fund and was performed in cooperation with the EIT ICT Labs Budapest Associate Partner Group.
This research is partially supported from the project POSDRU/88/ 1.5/S/60185 Innovative doctoral studies in a Knowledge Based Society PhD scholarship, Project co-financed by the SECTORAL OPERATIONAL PROGRAM FOR HUMAN RESOURCES DEVELOPMENT 2007 - 2013, Babeş-Bolyai University, Cluj-Napoca, Romania.

References

1. Akkoyunlu, E., Ekanadham, K., Huber, R.: Some constraints and tradeoffs in the design of network communications. ACM SIGOPS Operating ... pp. 67–74 (1975), http://dl.acm.org/citation.cfm?id=806523
2. Banavar, G., Chandra, T., Strom, R., Sturman, D.: A case for message oriented middleware. Lecture Notes in Computer Science, Distributed Computing 1693, 1–17 (1999), http://link.springer.com/chapter/10.1007/3-540-48169-9_1
3. Berman, S., Kesterson-Townes, L., Marshall, A., Srivathsa, R.: The power of cloud: Driving business model innovation. Tech. rep., IBM Global Business Services (2011), http://www.ibm.com/cloud-computing/us/en/assets/power-of-cloud-for-bus-model-innovation.pdf
4. Beverly Yang, B., Garcia-Molina, H.: Designing a super-peer network. In: Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405). pp. 49–60. IEEE (2003), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1260781
5. Birman, K.P., Joseph, T.A.: Exploiting virtual synchrony in distributed systems. In: 11th ACM Symposium on Operating systems principles. pp. 123–138. ACM New York, NY, USA (1987), http://dl.acm.org/citation.cfm?id=37515
6. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research (2002), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.7902&rep=rep1&type=pdf
7. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. Selected Areas in Communications 20(8), 100–110 (2002), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1038579
8. Curry, E.: Message-oriented middleware. In: Middleware for communications, chap. 1. John Wiley & Sons, Ltd, Chichester, UK (2005)
9. Girault, M., Cohen, R., Campana, M.: A generalized birthday attack. Advances in Cryptology - Eurocrypt '88 pp. 129–156 (1988), http://link.springer.com/chapter/10.1007/3-540-45961-8_12
10. Hintjens, P.: What is wrong with AMQP (and how to fix it). Tech. rep. (2008), http://www.imatix.com/articles:whats-wrong-with-amqp

11. IMatix Corporation: Multithreaded Magic with ØMQ. Tech. rep. (2010), http://zeromq.wdfiles.com/local--files/whitepapers%3Amultithreading-magic/imatix-multithreaded-magic.pdf
12. Innovations, R.T.: RTI Connext (2012)
13. Mell, P., Grance, T.: The NIST Definition of Cloud Computing, Recommendations of the National Institute of Standards and Technolog. National Institute of Standards and Technology (2011), http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
14. Oasis: Advanced Message Queuing Protocol Specification - Version 1.0. Tech. Rep. October (2012), http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
15. OMG: Data distribution service for real-time systems - Version 1.2. Tech. Rep. January (2007), http://kurser.iha.dk/ee-ict-master/timico/slides/2012_Fischer_DDS_Slides.pdf
16. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Middleware 2001 (November 2001) (2001), http://link.springer.com/chapter/10.1007/3-540-45518-3_18
17. Schneider, S., Farabaugh, B.: Is DDS for You? A Whitepaper by Real-Time Innovations (April) (2009), http://omg.org/news/whitepapers/IsDDS4U.pdf
18. Sústrik, M.: Broker vs. Brokerless. Tech. rep., iMatix (2008), http://zeromq.org/whitepapers:brokerless
19. Sústrik, M.: ØMQ: The Theoretical Foundation (2011), http://250bpm.com/concepts
20. Szilágyi, P.: ErlHop: Erlang Hosting Platform. Achieving Dynamic Scalability and Load Balancing on a Cloud Architecture. Masters, Babes-Bolyai University (2010), https://dl.dropboxusercontent.com/u/10435909/Documents/Theses/MSc-ErlangHostingPlatform.pdf
21. Szilágyi, P.: Decentralized bootstrapping in clouds. In: 10th Jubilee International Symposium on Intelligent Systems and Informatics. pp. 277–281. IEEE, Subotica (Sep 2012), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6339529
22. Szilágyi, P.: Securing communication in a peer-to-peer messaging middleware. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, Timisoara, Romania (2013)
23. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms (2nd Edition). Prentice Hall, 2 edn. (2006)
24. Vinoski, S.: Advanced message queuing protocol. Internet Computing, IEEE (December), 87–89 (2006), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4012603

Péter Szilágyi is currently finalizing his PhD studies at the Eötvös Loránd University in Budapest and Babeş-Bolyai University in Cluj, while also being enrolled as a business student of the European Institute for Innovation and Technology. At the moment he's focused on simplifying the development of backend services through the Iris project, and the launch of a public demo service on top of it, called RegionRank.

Received: August 23, 2013; Accepted: January 10, 2014.