Abstract— We demonstrate how enabling loosely-coupled communication mechanisms can allow powerful and easily accomplished collaboration between distributed systems. It is also shown how desirable properties such as scalability and fault-tolerance can be imparted to distributed systems by utilizing loosely-coupled mechanisms. Loose-coupling is briefly described and a frame-work (ChAMp) that achieves it through use of application-level multicast is detailed. A non-trivial distributed monitoring system (DeMon) that demonstrates the use of loose-coupling is described. DeMon, by itself is an important contribution as it allows scalable, fault-tolerant monitoring of large target sets.

Index Terms—Distributed Systems, Loose-coupling, Distributed Monitoring, Application-level Multicast, Dynamic-Metric Routing

I. INTRODUCTION

Use-cases of different distributed systems (not just different components of a single system) wanting to communicate between each other are not hard to come by. Imagine a scenario where a Content Distribution Network (CDN) wants to place some content close to where there is a demand for such content. The CDN knows what content is popular where, and it knows how to place that content in some accessible manner. Let us assume that there are a large number of Application Level Servers (used for various purposes, including that of the CDN) and that the CDN is just one of the customers of this 'Computing-Service' (CS) provider. Given the constraints of the CDN (including content popularity, location of audience etc) we would need to provide a set of servers from the (CS) pool. For this we would require a 'resource-discovery' (RD) mechanism which is able to provide a set of servers in some location given storage and processing-power requirements. We have shown here a scenario where three distributed systems (CDN, CS, RD) interact to provide services to the end user.

Both the CDN service and the Resource-Discovery (RD) service are distributed themselves - they are spread across the world. How does the CDN service know which host (service-implementer) to communicate with among the many hosts of the RD service? How does it even know the list of hosts of the RD service?

And there can be many other systems that these above mentioned services want to communicate with - i.e. there can be service composition between distributed systems. To enable this service composition in a generic way, so that every pair of 'composing' services does not need to invent a method of discovery and communication, we propose a standard 'channel' based mechanism. A 'channel' here is defined by the content it carries, not by the end-points it connects. Our argument for this is similar to the argument making the case for the need for middleware: why re-invent the wheel each time?

Distributed systems collaborate by tuning-in to different channels to receive information based on their needs. And other systems routinely place information on the channel based on the function they perform. This mechanism allows systems to collaborate without knowing complete details about each other. All they need to know is the details of the channel and format of information on the channel (which they need to know anyway to interpret it). This model of collaboration is an example of loose-coupling.

The goals of this paper are three-fold. Firstly, we explain the loose-coupling model and the useful properties that it affords distributed systems. Secondly, we show how to build a framework that enables loose-coupling by describing ChAMp (Channel based Application level Multicast Protocol) – our application level multicast routing prototype, which is based on emulation of IP multicast. Lastly, but importantly, we
DeMon (DistributEd Monitor) – a novel and original distributed monitoring application that allows scalable and fault-tolerant monitoring of large target sets. DeMon also demonstrates how loose-coupling enables its scalability and fault-tolerance. The big picture consisting of DeMon, ChAMp and loose-coupling is shown in Fig. 1.

II. LOOSE-COUPLING

Loose-coupling changes the standard convention of a client initiating a connection to a service and waiting for a response. Instead a service publishes events to a communication channel with no a priori knowledge of the clients that are subscribed. The communication channel isolates a ‘service’ from a service instance (an implementer of the specific service)[2]. Many of the advantages of loose-coupling come from the fact that systems do not need to have complete knowledge of each other to collaborate. Loose binding of components enables,

i) Transparent Migration of service instances: Implementers of a service are allowed to move to different locations without adversely affecting their consumers and even without the knowledge of consumers.

ii) Transparent Replication of service instances: Because consumers are not directly coupled to implementers, there can be transparent replication. For example as load increases portion of the services can be off-loaded to another service-implementer without requiring the consumer to perform a hand-off/on.

iii) Simplifies composition among services: Composition and collaboration between many systems is complicated because of the added knowledge that must be encoded about the specific behaviors of each of the systems one wishes to collaborate with (e.g. knowledge of hand-off mechanisms required when replication or load-balancing occurs). By making many of these aspects transparent (see above points) composition of services is simplified.

iv) Hides differences among multiple implementations of a particular service: This is similar to the object-oriented concept of separating the implementation from interface (information hiding; what you don’t know will not hurt you). As long as service implementers place information in a format understandable by consumers, how they obtain or compute that information becomes less important.

The above mentioned features can also be viewed as independence properties of the loosely-coupled system such as a) Location (endpoint) independence – Services rendezvous with each other without needing to know the location of particular instances of the service; b) Service independence – Neither the publisher nor the subscriber of a service needs to know of the existence of the other; and c) Timing independence – The most basic form of communication among components of a service is asynchronous [2]. We next detail a solution to creating these ‘channels’.

III. APPLICATION-LEVEL MULTICAST

Our approach is to view this ‘channel’ abstraction as a multicast problem. We prototype an application-level-multicast as an infrastructure service (like the IP-multicast is an infrastructure service) at the Application level (versus the non-existent IP-multicast being at the network-layer). It is important to note that there is no real underlying IP multicast available, even after over a decade of research and implementation. And hence Application-level multicast solutions have become the paradigm of choice.

There are two broad categories of Application-Level Multicast (ALM) solutions – proxy-based and peer-to-peer based architectures. In a peer-to-peer architecture, all functionality required to implement multicast is
placed at the end hosts actually participating in the multicast group. Such architectures are thus completely distributed with each end host maintaining state only for those groups it is actually participating in. In a proxy-based architecture, an organization that provides value added services deploys proxies at strategic locations. The functionality of multicast is only placed at the proxies. End hosts attach themselves to proxies near them, and receive data using plain unicast, or any available multicast media [1]. Peer-to-peer architectures typically serve specific functionality (such say an implementation for streaming multimedia content). Proxy-based architectures are meant to serve diverse groups and are more generic. They are also required to serve a large number of groups (due to their generic nature). The size of the individual groups or that of the entire user set in both systems may be the same. As can be seen in the next section, we take the approach of proxy-based architecture and the role of a router is distinct from that of the end-host.

IV. CHAMP: CHANNEL-BASED APPLICATION LEVEL MULTICAST PROTOCOL

A. Emulation of IP-Multicast & Packet types

ChAMPp provides ALM by partially emulating the protocol and semantics of IP multicast (IPM). As will be seen from its design, it is also different in some aspects from IPM. The ‘channel’ in CHAMP correlates with the multicast-group in IPM. Just as in IPM, senders to channels do not need to formally join the channel; any host connected to CHAMP can send to any channel that it wishes (even if that channel has no listeners). An end-host wanting to receive information over a channel, tunes-in to the channel by sending a ‘join’ message to its immediate gateway (the gateway is the end-hosts interface to CHAMP – proxy-based architecture). On a successful ‘join’, an end-host is given a lease on the channel, which is the period of time for which it can receive information. The end-host is free to renew its lease whenever it requires (it needs to periodically do so to remain connected). When an end-host no longer wishes to receive information it may send a ‘prune’ message asking to be disconnected from a particular channel. The lease mechanism is in place to support soft-state at the routers so that router state doesn’t grow and explode when nodes do not remove themselves cleanly from channels.

B. Protocol

Every router on startup is provided with a neighbor-set. How neighbors are chosen (either at startup or dynamically throughout life of router) is not specified. It is assumed here that a topology is known. We detail the multicast protocol as a set of actions to be performed in reaction to events. As can be seen it resembles IPM very closely.

- On reception of a ‘join’ message:
  a. If message is from a client (all addresses not of neighbors are assumed to be clients): If new channel is new, then create state for new channel. Add the client as a subscriber to the channel. Then flood the join message to all neighbors (after changing src-address to current router’s address).
  b. If message is from neighbor: Then check if that neighbor is on the shortest-path from current router to the original source of the join message (i.e. the original client or its gateway router). If yes, then process the message by adding the neighbor as a subscriber of the channel. If we haven’t flooded a join message for the same channel recently then flood the join message to the remaining neighbors. But if neighbor is not the next-hop on the shortest path, then just drop the message. This check is called Reverse-Path-Forwarding or RPF (see next sub-section for details).

- On reception of a ‘data’ message:
  a. If message is from a client: If current router knows any subscribers (clients or neighbors) for this channel, then it forwards each of them the data message.
  b. If message is from neighbor: If there are any end-hosts (clients) who are subscribers of this channel, then forward the data message to each of them. Perform a reverse shortest-path check on the neighbor that forwarded the message to the current router. If the check fails, then drop the message. If the check passes and if there are neighbors that are subscribers of this channel, then forward the message to each one.
- On reception of a ‘prune’ message:
  a. If message is from a client: Remove the client (if it is a subscriber) from the channel’s subscriber list. If the subscriber list for this channel is empty (i.e. no neighbors or clients), then forward the prune message to all neighbors.
  c. If message is from neighbor: If the neighbor was a subscriber of the channel then remove then neighbor from the subscriber list. If our subscriber list became empty because of the removal of the neighbor, then create a new prune message and forward it to all neighbors (except the one that sent it to us).

Clients need to periodically renew their subscription by sending join messages to their gateway routers.

C. Reverse Path Forwarding
Reverse-path forwarding (RPF) is an important part of keeping the flooding involved in multicast under control, taking care not to generate duplicate packets and making sure the flood terminates. It also makes sure that packets traverse the best possible routes relative to the source. Under RPF a packet is forwarded only under one condition – if the packet is from a neighbor who is on the current router’s next-hop on the shortest-path to the original source of the message. This allows the creation of single-source shortest-path distribution trees just with the state kept by a standard unicast distance-vector protocol, i.e. given a destination, what is the best next-hop (i.e. next hop on the shortest path to that destination). Therefore to do RPF (and hence keep multicast floods in check) we need an underlying unicast protocol. We detail that next section.

D. Design, Architecture & Implementation
ChAMp is built over Java - Remote Method Invocation (RMI). Java RMI provides some useful facilities that ChAMp does not need to reinvent such as interpreting an address-space (which ChAMp can view in an opaque fashion), providing serialization of java message-objects (messages implemented as java classes implementing the Serializable interface) and dealing with all the issues related to network communication. One of the disadvantages of using RMI is that the method-invocation model provides blocking semantics, whereas ChAMp requires a non-blocking message-passing mechanism. If ChAMp (or any router for that matter) uses blocking-invocation to route packets, then it would suffer from a serious case of distributed-deadlocks (because state in the routers is protected by locks). To avoid this problem we first built a Generic-Queue-Processor that provides a non-blocking message-passing interface over RMI.

Why not Jabber Technology? We had experimented with the use of Jabber [5] for an earlier project (which first demonstrated some ideas related to loose-coupling by building a framework over jabber). But jabber had two important disadvantages; its was (then and probably is even now) still an unstable technology and implementation, causing frequent interruptions in solving jabber related issues; jabber is a xml-based instant messaging solution and all content is encoded into (and hence needs to be decoded from) xml causing significant performance issues.

Another misconception about the Jabber technology is that it offers some sort of magic-multicast solution. In fact, the only multicast facility jabber provides is a rudimentary chat-room that keeps a list of subscribers and sends all traffic in the room to all members of that chat-room. This trivial single-server list-based
implementation maybe suitable for chat-rooms of small to medium sizes, but definitely does not scale to any non-trivial network routing requirements. Instead one must use the primitives provided by the Jabber system to build multicast (much like we use the services provided by RMI to build ChAMP). Often the chat-room abstraction is thought to be some multicast-panacea and used as such. But Jabber technology is promising and with more stability and users’ understanding of it, it may become far more usable.

ChAMP’s Remote Interfaces: The Generic-Processor (which sits below the actual routing functionality) exports two remote interfaces: RemoteRouter interface and the RemoteQueue interface. These are the only two remote interfaces in the entire ChAMP (and DeMon) system. An entity (client or router) wishing to send a packet places the packet on its local outgoing-queue. A queue-processor routinely samples the outgoing-queue and is in-charge of the actual transfer. It first acquires a reference to the RemoteRouter interface of the destination. It then makes a getRemoteQueue() call on that interface, which returns a RemoteQueue interface reference. Then one can call enqueue(packet) on the RemoteQueue. This places the message on the destinations incoming-queue. The incoming-queue is managed locally by a thread that periodically removes the 1st message and calls the router with the message. We next describe the unicast protocol, which is the standard Path-Vector (Bellman-Ford) protocol with an addition to support dynamic-metrics.

V. Unicast Service with Dynamic-Metrics

The Unicast routing component is crucial to providing the multicast described above as it allows RPF (reverse-path forwarding). In other words, some how a way to provide all-pairs shortest paths knowledge in a distributed fashion is required and the unicast-routing using the standard Distance-Vectorizer(DV) routing protocol, based on the Bellman-Ford algorithm, is one well used method.

We implement a Path-Vector protocol, which is a modification of the DV protocol to avoid routing loops. BGP (Border Gateway Protocol) [4] is a popular protocol that uses Path-Vectors. Our unicast protocol is a simplified version of BGP, in that it does not allow any role for policy in routing (whereas BGP allows policy specification). The unicast protocol is built over the same Generic-Processor described earlier (in the multicast section). We omit details of the actual protocol as it is just an implementation of the well known Bell-Ford algorithm. Instead we turn to looking at the case of dynamic-metric routing.

A. Dynamic-Metric Routing & Scout

Standard-DV is not a dynamic metric-based routing algorithm but relies on static link costs. This means that standard-DV does not automatically choose a 'best' route based on some dynamic metric (such as the bandwidth available over that link). Because of this it is not able to react to a class of non-stationarity in the network, namely dynamically changing link costs (e.g. due to congestion).

Why cannot DV use dynamic-metrics in-place of static link costs? i) Routing with DV using dynamic metrics can cause oscillations – where all traffic in a congested zone is shifted at the same time to an uncongested zone (potentially leading to congestion in the new zone). This is because routes to every destination are affected if a link on the path to that destination changes its cost. And ii) the cost of a link based on a dynamic-metric is dependant on the data-traffic currently on the link. Since routing update frequencies depend on changes, the nature of the data-traffic dictates the amount of routing updates. This means that it is hard to control the amount of routing traffic.

We study Scout[3] which is a Destination-initiated shortest path routing technique based on the observation that there is a significant amount of locality in traffic - a lot of traffic goes to few destinations. Scout solves some of standard-DV’s dynamic metric problems. As Scout is destination-initiated, a destination triggers the route computation to itself at periodic intervals. So the destination controls the frequency of recomputations. The routing overhead is independent of the data-traffic in the network, hence is predictable and controllable by initiator – solving the 1st problem identified above. Updating of routes to different destinations is uncorrelated, as the destinations independently (at their required frequencies) trigger updates. But in traditional algorithms (DV, LS) cost changes cause all routes (to all destinations) to be affected, causing oscillations.
Because its destination initiated Scout cannot make use of the efficiency that comes from aggregating information about all destinations (which standard-DV uses). Therefore the flexibility of using dynamic metrics in Scout comes at the price of much higher routing-control traffic. But since only destinations that are hot-spots need to perform Scout-based routing, this cost is mitigated. We next detail some Scout terminology and then use that to provide a listing of the Scout protocol.

B. Scout Protocol

Terminology:

i) Scout Message – a scout is a message of the form \([R, C_R, Seq]\), where \(R\) is the node that initiated the scout, \(C_R\) is the cost to \(R\) from current point in network (initially \(C_R\) is zero) and \(Seq\) is the sequence number that is sent with every scout and incremented by the initiator.

ii) Broadcast-Interval (BI) – The interval between successive floods by the initiator, i.e. the inverse of frequency of sampling.

iii) Designated-Neighbor – Node P’s designated-neighbor to R in the current-BI is the neighbor that gave P the least-cost scout to R in previous-BI.

Protocol:

1) Destinations periodically generate a Scout with increasing sequence number.

2) On receiving a Scout, router discards Scout if:

   a) Scout sequence number is not current or

   b) The Scout advertises path to current node

3) Router updates Scout by adding cost of incoming link to cost in Scout and

   a) If Router has not forwarded a Scout (from same source) in the last BI, flood the Scout

   b) Else if Scout is from designated neighbor, forward least-cost Scout (from same source) received in current BI.

   c) Else store the Scout.

4) Update forwarding table to reflect the shortest path.

C. Evaluation Methodology:

We used a simple topology consisting of 5 routers. Of these 5, scouter_05 runs the Scout-DV hybrid. The other four routers run the standard-DV protocol. In this topology the Dst01 is the destination that is receiving traffic from both Src01 and Src02. As can be seen in Fig.3 the static shortest-path between Src01 and Dst01 is Router_1-2-4-5 and that between Src02 and Dst01 is Router_4-5. The portion Router_4-5 is common between the two paths. When both the sources are sending traffic this path is congested. Note that there is an alternate sub-optimal (in terms of static link cost) path between Src01 and Dst01.

To be able to evaluate the Scout-DV hybrid, a way to estimate the true cost of a link (i.e. dynamic cost) is needed. We used the length of the incoming queue as a measure of the congestion on the link. If the queue is always empty then there is no congestion and there is no extra cost added due to queuing. When queuing occurs we add this cost to the static link cost (or the base minimum cost) to obtain a dynamic-link cost.

\[\text{Dynamic-link cost} = \text{static-link cost} + \text{length of incoming queue}\]

When a data message is forwarded over a particular link, the dynamic-cost of that link is added to the
existing cost of the message. In this way when a message arrives at the destination, the Dst01 is able to tell the cost expended on the message. The evaluation-metric, then, is the average-per-message-cost.

D. Results:

Hypothesis: Scout-DV Hybrid is able to effectively shift selected (i.e. destination-wise) traffic from a congested-zone to uncongested-zone. To check this we can look at the Average Message cost as calculated by the Dst01 for all messages it receives. We perform this once for a purely standard-DV based network and then for a network where scouter_05 is running Scout. We also look at the traces of the log files to examine the paths that a message takes (we keep a traceroute like path-vector in each message).

In a lightly loaded Network: Both sources are sending at a low rate of 1 message per second. The queue-managers working at about the same frequency are able to effectively handle all traffic without any queuing. This means that there is no congestion in the network. In this case the Scout-DV Hybrid performed comparably with Standard-DV providing an average-message-cost = 6. (This is nothing but the simple average of the two least-cost path lengths from Src01 and Src02).

In a heavily loaded Network: Both sources are sending at a high rate of 10 messages per second. The queue-managers are processing messages at 1/10th that rate, leading to formation of large queues. From the traces we could observe clearly that Scout-DV moved traffic from Scr01 through Router_01—Router_03—Router_05 path, instead of the other more congested path. Standard-DV provided average-message-cost=18. And Scout-DV Hybrid provided average-message-cost=15, an improvement of 20%.

E. Poor Performance with Multicast:

When the above experiment was conducted with multicast traffic we observed very poor performance. In particular we saw two types of problems: i) High Loss and/or ii) High duplication rates. To understand why there is such a big difference between the unicast & multicast cases, one must take a closer look at Reverse-Path forwarding used in multicast.

In unicast routing if the state in all the routers has not converged (due to topological changes such as link failures or link-revivals), then a packet may get routed in a circuitous fashion or may loop until the network state converges (or the TTL expires). So unstable network images in unicast-routers do not immediately lead to loss, but can lead to higher delays (though prolonged instability can lead to loss due to TTL expiry on packets).

In multicast, on the other hand, RP Forwarding says that if the neighbor router who forwarded the packet is not on the shortest path to the source, then the packet should be dropped immediately. This is done in the hope that the packet will be correctly forwarded by the routers in the true-shortest path. Two possible scenarios (out of many) are that either both the routers (considering two, but there can be some k) think they are on the shortest path and forward the packet (causing duplication) or both think that they are not on the shortest path, causing loss. Both are detrimental to the network.

The question to ask is why the routers’ state is not converging. Due to the use of dynamic metrics, changes in link costs in the network affect the convergence of router state. If the rate of change in the network (i.e. of link costs) is high enough, then convergence cannot take place, causing perennial duplication and/or loss of multicast packets.

Possible Solution ⇒ Hysterisis: One possible solution may be (we have not tried the solution, so can
only conjecture as to its efficacy) to use **hysterisis**. Hysteresis is a damping mechanism, wherein reaction to change occurs only if the magnitude of change is above a certain threshold. For example, if we had set a threshold on how much costs must differ for a router to replace a costlier route with a cheaper route that would be introduction of hysteresis. This could in theory bring down the rate of perceived change (one cannot bring down the true rate of change) to a level that would allow convergence. Of course the amount of hysteresis is a trade-off between how responsive to change one wants the network to be versus, how fast one would want the state to converge. The rest of the project uses only the **static-cost standard Path-Vector unicast protocol and the multicast protocol above that**.

VI. **DeMon: Distributed Monitor**

It has been demonstrated how to build a framework for loose-coupling. Now we will put it to use by building a novel Distributed Monitoring service which is **scalable, fault-tolerant** and particularly tries to avoid the implosion problem that can be experienced by monitors observing large target sets (we use the terminology ‘target-set’ to refer to the set of hosts/resources being monitored). A very important property of DeMon is that it tries very hard to avoid explicit network communication for the sake of coordination among its components, i.e. it doesn’t need to be aware of the live-ness of the individual components and also doesn’t have any significant control overhead of its own. These two properties influence its protocol heavily.

DeMon (or DistributEd MONitor) has two main types of components: i) Sensors and ii) Monitors. Sensors are entities which live on (or have direct access) to the systems being observed. Sensors periodically generate information about the system that they are in-charge of. In this paper it is assumed that there is at least one sensor per resource (but this assumption does not drastically affect any functionality). Monitors are remote entities which are interested in the information that sensors periodically generate. There are two types of Monitors in DeMon – sub-monitors and main-monitors. Main-monitors are the ultimate recipients/consumers of DeMon, whereas sub-monitors are intermediate monitors known only to the DeMon implementation. We will shortly see what the exact role of a sub-monitor is.

We also make some assumptions about the knowledge of the namespace of sensors. It is assumed that there is complete knowledge of the namespace by all sensors and that all sensors have unique names. It is also assumed that the number of sensors (and their types) is also known. Importantly we don’t make the assumption regarding knowledge of live/dead sensors, just the total number of sensors.

In our proposed (and implemented) solution all the sensors and monitors (both main-monitors as well as sub-monitors) are clients of ChAMp. Sensors are information generators in ChAMp, i.e. they place information periodically on channels. Main-monitors are consumers of information i.e. they ‘join’ a channel and are given any information that arrives on a channel. Sub-monitors perform both operations, receiving on channels and multicasting on (typically different) channels.

A. **DeMon Protocol:**

We now detail the protocol used by sensors and sub-monitors in DeMon to create a scalable, fault-tolerant environment for any entity which wishes to receive the generated sensory-information (i.e. consumers, also called as man-monitors).

1. Every sensor is aware of its own name. It uses a deterministic-function of its name and functionality to create **a unique name for its own channel** (termed a personal-channel henceforth) on ChAMp. E.g. A sensor named Sensor01 which generates UNIX-top like data can create a **channel named top.01** over ChAMp. This deterministic-function is known by all other sensors in the system as well, so that by knowing another sensor’s name and function, they can map to its personal channel.

2. At startup, **some** sensors also become monitors. These are the monitors which are termed **sub-monitors**. When we say ‘some’ we mean that there is a mechanism to decide on startup if a sensor should instead be a sensor + sub-monitor. In DeMon this mechanism is uniform random selection of sensors to become sub-monitors - i.e. on startup a sensor generates a random number and based on that value decides on its own (without requiring network communication with other components) to become a sub-monitor. For example, say if condition to become a sub-monitor is ‘p < 0.1’, where p is the randomly
generated number, then there is one-in-ten chance that a sensor becomes a sub-monitor. This should lead on average to 10% of the sensors becoming sub-monitors.

3. The sub-monitors (those which were spawned in the previous step) now will need to

a) ‘pick’ some subset of the Sensor namespace,

b) map those names to the names of the personal channels of those sensors and then

c) join those personal channels, by sending join messages to ChAMP

The mechanism to ‘pick’ the subset of sensors (as mentioned in point 3.a above), in DeMon, is to again take recourse to uniform-randomness. The sub-monitors individually sample the entire sensor namespace in a uniform random fashion. Out of that sampling they pick some N sensor channels to join. For example a sensor named sensor01 may have become a sub-monitor and it could pick sensors Sensor03 and Sensor05 (and hence their respective channels top.03 and top.05 to join). Other methods to pick could be the use some type of deterministic mapping from sub-monitor ids to the sensor-namespace. This is very similar to a DHT, where an owner is responsible for specific namespace regions. In DeMon no single owner is responsible for any specific region. Due to the uniform randomness, there is equal probability of any sub-monitor representing (i.e. picking to listen on) any sensor’s personal channel.

4. Once the sub-monitors join the chosen channels, they start to receive information placed by the individual sensors. Because the different sub-monitors only receive a relatively small portion of the sensor traffic each (depending on the number of sensors and the value of p in step 2), they are not bombarded with more information than they can handle. Another advantage afforded by this loose-coupling and channel mechanism is that a type of open-loop flow-control can be performed in extreme circumstances by which a sub-monitor can tune-out (i.e. send a prune message) to some channels if it cannot handle the aggregate load.

5. Sub-monitors aggregate this received information in some form by applying a transform. The simplest transform could be the application of union over the received information. Other useful transforms could also be performed (for e.g. in our implemented distributed ‘top’ sensor over DeMon, the sub-monitors add up the total resources used by different instances of the same process on the same hast – if there are 4 java processes running, the sub-monitors collapse that by adding together the 4 cpu usages.).

6. Sub-monitors periodically (usually of a period greater than the rate at which sensors put out information, so as to allow more aggregation) place the aggregated information on the ‘real’ channel. For example, in the case of the D-Top facility, the personal-channels were named ‘top.01’, ‘top.02’….. and the real-channel was named ‘top’. So the sub-monitors will listen on top.01 and top.03, for instance, and place aggregated information on channel top.

7. The consumers or main-monitors listen on the real-channels and receive the aggregated information. In particular the rate of information arriving and the amount of information arriving is far lesser than that of directly monitoring each sensor. We expect that there is at least an order of magnitude difference in the number of packets that a main-monitor receives under DeMon than under direct monitoring of all sensors. This allows monitoring of very large target-sets.

B. Issues with DeMon Protocol:

As mentioned earlier, DeMon tries to provide scalability and fault-tolerance without having to use explicit coordination mechanisms between sensors, sub-monitors and main-monitors. Most decisions are made probabilistically. The issues are primarily with the ‘picking’ of sensors by sub-monitors to listen to. Because of the uniform random sampling of the sensor-namespace two issues can arise – Gaps and Overlaps.

Gaps are portions of the sensor namespace that no sub-monitor is listening to. This means that due to the lack of representation of those portions of the namespace, the sensory data of those hosts/resources are
not being aggregated and multicast on the real-channel. That means that consumers may not receive complete information of the target-set even if all the sensors on the target set are alive and well.

Overlaps are portions of the sensor namespace that more than one sub-monitor listen on. This means that there is redundant information being propagated within ChAMp and that the end-consumers (main-monitors) will receive redundant information and must be prepared to handle the same (i.e. there must be a mechanism to identify duplicate information – as simple as sending the address of the original sensor, so as to match and reject duplicates).

Gaps are definitely problems and can only be detrimental to the system (since portion of the target-set is unobservable, bad things can happen unseen!). But overlaps, on the other hand, have some beneficial effects as well. In particular, the redundancy they afford is a natural fault-tolerant mechanism against the failure of sub-monitors or even portions of ChAMp. We now see how we can handle the problem of Gaps.

C. Possible Solutions to Gaps & Overlaps:

Solution 1 \(\rightarrow\) Rounds: Both problems (Gaps & Overlaps) occur due to the uneven representation of the sensor namespace. If the unfairness is bounded by some manner, then another trial at representation can be allowed. It is most probable that even this second representation will be uneven, but it is also probable, due to the uniform random nature of sampling, that the unevenness of the representation should definitely be different. The idea is to formalize this notion of uneven representations within bounded intervals. We call these intervals Rounds. Instead of 'picking' sensors from the namespace to listen to only at startup, the sub-monitors perform the picking at the beginning of every round. It is hoped that over some number of rounds (k, we have not performed empirical evaluation to ascertain likely k) there will be even representation of the namespace. This doesn’t guarantee that there is always even representation (which is impossible without determinism and/or network coordination) but that there will be even representation over some period. Rounds are an elegant way to solve the unfairness problem.

Solution 2 \(\rightarrow\) Hybrid Approach: This second solution addresses only the problem of Gaps (unlike Rounds which addressed both problems). Here we suggest a different (and orthogonal) change to the DeMon protocol. Currently Sensors only send on their personal-channels and sub-monitors send on the real-channel. We suggest that at relatively low-rate (as compared to the rate at which Sensors place information on personal-channels), Sensors also place information on the real-channels. This guarantees that that there is complete information available, if not always, at least at low-rates.

Both the above mentioned solutions can be implemented together as their mechanisms and resulting improvements are orthogonal in nature.

D. What Loose-coupling does for DeMon?

We had earlier expounded the virtues of loose-coupling; we now show how DeMon-over-ChAMp demonstrates those virtues. Infact DeMon’s design without performing explicit coordination between its components is possible because of the loose-coupling afforded. In a tightly coupled system one would need to be able to identify explicitly which and where the sub-monitors were. Collaboration with (or consumption of) DeMon is as simple as listening on a ‘real-channel’. If really required (for some reason) one could even listen on a personal-channel (if one knew how to address the channel). Faults of sub-monitors in the system may cause (in the absence of any redundancy) brief delays in information from a particular sensor. This is the only effect of faults that is observable by a client. In the next Round another sub-monitor(s) may represent the sensor. And lastly the scalability is offered in a purely transparent fashion to clients due to the loose-coupling.

VII. Conclusion

It has been demonstrated how to build a framework to provide loosely coupled channel-based communication through a prototype Application-level multicast protocol ChAMp. ChAMp is largely based on existing protocols (both IP multicast & unicast). Its implementation over RMI is original work. A scalable, fault-tolerant Distributed Monitoring system, DeMon, has also been built and demonstrated. DeMon (unlike ChAMp) is based completely on original ideas and is as such original work, both conceptually and implementation-wise. DeMon also demonstrates how powerful loose-coupling can be and what facilities it can afford distributed systems. In future, we wish to completely implement the DeMon protocol and evaluate it with fairly large target-sets. We wish to especially see how its stochastic decision making scales and how our Rounds-based and Hybrid-based solutions to problems of Gaps & Overlaps fare.
REFERENCES


