Using Gossip Protocols for Failure Detection, Monitoring, Messaging and Other Good Things

Monday, November 14, 2011 at 9:15AM

Todd Hoff in Paper, Strategy, gossip

When building a system on top of a set of wildly uncooperative and unruly computers you have knowledge problems: knowing when other nodes are dead; knowing when nodes become alive; getting information about other nodes so you can make local decisions, like knowing which node should handle a request based on a scheme for assigning nodes to a certain range of users; learning about new configuration data; agreeing on data values; and so on.

How do you solve these problems?

A common centralized approach is to use a database and all nodes query it for information. Obvious availability and performance issues for large distributed clusters. Another approach is to use Paxos, a protocol for solving consensus in a network to maintain strict consistency requirements for small groups of unreliable processes. Not practical when larger number of nodes are involved.

So what's the super cool decentralized way to bring order to large clusters?

Gossip protocols, which maintain relaxed consistency requirements amongst a very large group of nodes. A gossip protocol is simple in concept. Each nodes sends out some data to a set of other nodes. Data propagates through the system node by node like a virus. Eventually data propagates to every node in the system. It's a way for nodes to build a global map from limited local interactions.

As you might imagine there are all sorts of subtleties involved, but at its core it's a simple and robust system. A node only has to send to a subset of other nodes. That's it.

Cassandra, for example, uses what's called an anti-entropy version of the gossip protocol for repairing unread data using Merkle Trees. Riak uses a gossip protocol to *share and communicate ring state and bucket properties around the cluster*.

For a detailed look at using gossip protocols take a look at GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems by Rajagopal Subramaniyan, Pirabhu Raman, Alan George, and Matthew Radlinski. I really like this paper because of how marvelously well written and clear it is on how to use gossip protocols to detect node failures and load balance based on data sampled from other other nodes. Details are explained clearly and it dares to cover a variety of possibly useful topics.

From the abstract:

> *Gossip protocols have proven to be effective means by which failures can be detected in large, distributed systems in an asynchronous manner without the limitations associated with reliable multicasting for group communications. In this paper, we discuss the development and features of a Gossip-Enabled Monitoring Service (GEMS), a highly responsive and scalable resource monitoring service, to monitor health*

and performance information in heterogeneous distributed systems. GEMS has many novel and essential features such as detection of network partitions and dynamic insertion of new nodes into the service. Easily extensible, GEMS also incorporates facilities for distributing arbitrary system and application-specific data. We present experiments and analytical projections demonstrating scalability, fast response times and low resource utilization requirements, making GEMS a potent solution for resource monitoring in distributed computing.

Failure Detection

The failure detection part of the paper is good and makes sense. By combining the reachability data from a lot of different nodes you can quickly determine when a node is down. When a node is down, for example, there's no need to attempt to write to that node, saving queue space, CPU, and bandwidth.

In a distributed system you need at least two independent sources of information to mark a node down. It's not enough to simply say because your node can't contact another node that the other node is down. It's quite possible that your node is broken and the other node is fine. But if other nodes in the system also see that other node is dead then you can with some confidence conclude that that node is dead. Many complex hard to debug bugs are hidden here. How do you know what other nodes are seeing? Via a gossip protocol exchanging this kind of reachability data.

In embedded systems the backplane often has traces between nodes so a local system can get an independent source of confirmation that a given node is dead, or alive, or transitioning between the two states. If the datacenter is really the computer, it would be nice to see datacenters step up and implement higher level services like node liveness and time syncing so every application doesn't have to worry about these issues, again.

The paper covers the obvious issue of scaling as the number of nodes increases by dividing nodes into groups and introducing a hierarchy of layers at which node information is aggregated. They found running the gossip protocol used less than 60 Kbps of bandwidth and less than 2% of CPU for a system of 128 nodes.

One thing I would add is the communication subsystem can also contribute what it learns about reachability, we don't just have to rely on a gossip heartbeat. If the communication layer can't reach a node that fact can be noted in a reachability table. This keeps data as up to date as possible.

Using Gossip as a Form of Messaging

In addition to failure detection, the paper shows how to transmit node and subsystem properties between nodes. This is a great extension and is a far more robust mechanism than individual modules using TCP connections to exchange data and command and control. We want to abstract communication out of application level code and this type of approach accomplishes that.

It seems somewhat obvious that you would transmit node properties to other nodes. Stats like load average, free memory, etc. would allow a local node to decide where to send work, for example. If a node is idle send it work (as long as everyone doesn't send it work at the same time). This local decision making angle is the key to scale. There's no centralized controller. Local nodes make local decisions based on local data. This can scale as far as the gossip protocol can scale.

What goes to another level is that they use an architecture I've used on several products, sending subsystem information so software modules on a node can send information to other modules on other nodes. For example, queue depth for a module could be sent out so other modules could gauge the work load. Alarm information could be sent out so other entities know the status of modules they are dependent on. Key information like configuration changes can be passed on. Even requests and response can be sent through this mechanism. At an architecture level this allows the aggregation of updates (from all sources on a node) so they can be sent in big blocks through the system instead of small messages, which is a big win.

This approach can be combined with a publish/subscribe topic registration system to reduce useless communication between nodes.

Another advantage of this approach is data could flow directly into your monitoring system rather than having a completely separate monitoring subsystem bolted on.

In the meat world we are warned against gossiping, it's a sin, it can ruin lives, it can ruin your reputation, etc., but in software, gossiping is a powerful tool in your distributed toolbox. So go forth and gossip.

Related Articles

- Gossip Girl
- ACID versus BASE for database transactions by John D. Cook
- Distributed Storage Systems by Swaroop C H
- The promise, and limitations, of gossip protocols by Ken Birman

- Phi Accrual Failure Detection by Naohiro Hayashibara, Xavier Défago, Rami Yared and Takuya Katayam. *Accrual failure detection is based on two primary ideas: that failure detection should be flexible by being decoupled from the application being monitored, and that outputting a continuous level of "suspicion" regarding how confident the monitor is that a node has failed.*
- Level-triggered and edge-triggered by Mike Burr
- A Gossip-Style Failure Detection Service by Robbert van Renesse, Yaron Minsky, and Mark Hayden
- Rumor Routing Algorithm for Sensor Networks by David Braginsky , Deborah Estrin
- Presentation Schedule // CS 525: Advanced Distributed Systems // Spring 2011
- Efficient On-Demand Operations in Large-Scale Infrastructures by Ko, Steven Y.
- Optimizing Information Flow in the Gossip Objects Platform by Ymir Vigfusson, Ken Birman, Qi Huang, Deepak P. Nataraj
- Paper: Consensus Protocols: Paxos
- History of Epidemics by Werner Vogels
- Epidemic Computing at Cornell by Werner Vogels
- Cornell Paper Directory by someone who likes really small fonts
- A Gossiping Protocol for Detecting Global Threshold Crossings
- Efficient Reconciliation and Flow Control for Anti-Entropy Protocols by Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. *Anti-entropy protocols can process only a limited rate of updates, and proposes and evaluates a new state reconciliation mechanism as well as a*
  ow control scheme for anti-entropy protocols.