Capacity Planning in Distributed Systems using Network Flow

One of the key challenges in building a scalable, distributed system is determining the capacity of the system and ascertaining cheap and efficient ways of increasing this capacity. At FlexGo (http://www.microsoft.com/flexgo), we build a scalable message-based distributed system in the cloud that clients and partners can connect to and work with. The taxonomy "client" and "partner" is used to differentiate the type of message incoming from a source, and can be generalized to supporting n incoming sources, each with possibly m distinct messages.

The necessity for assessing and analyzing the capacity of the system is justified by the following two reasons— determining where the bottlenecks in the system are, and providing a guarantee for service level agreements (SLA's). Knowledge of the bottlenecks in the system helps with determining efficient ways to scale-out and address these bottlenecks. While this may seem obvious as just looking for the node with the lowest capacity in the system, it's not as easy when the system looks like an interconnected graph. SLA's help clients/partners set expectations on how many messages they can send into the system with some guarantee of serviceability within a time interval. This can also be translated into the overall service time for each message at a specified current workload, but we will stick to the former for simplicity.

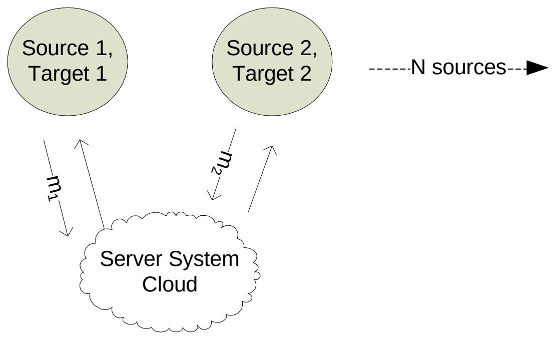


Figure 1

A simple depiction of this problem is provided in figure 1. There are N sources, each of which is capable of providing m_N (possibly distinct) requests into the system. The goal is to provide some guarantee to each of the sources that the system can process at least m_k messages from each source, assuming they all provide m_k and no more than m_k sources.

The easy solution to the problem that is done a lot in practice is to just test the real distributed system and look at the results. But this paper explores the use of network flow and graphs to come up with an alternate method.

The first step is to represent the distributed system as a graph. Internally, the distributed system is just a collection of services communicating through messages. This service-level framework can be represented as a graph with the nodes being the services, and the edges connecting services that pass messages between each other.

Perhaps a more accurate model is to represent this system as a topological (network) layout of the services, where the nodes correspond to physical units hosting each service and the edges still connect to services that communicate with each other. This model allows us to

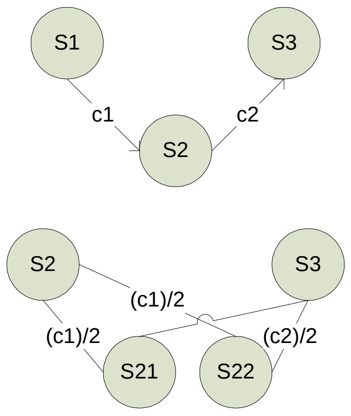


Figure 2

factor in articulation points, where if one physical unit fails, the flow of messages through this graph is affected. Figure 2 depicts this difference. The top model is where S2 is internally managed as several machines with total capacity c1. The bottom model shows S2 being represented as actual physical machines with each having half the capacity of the top one. If S21 dies, the capacity of S2 is now half of what it used to be.

It is hard to formalize and analyze a model without looking at the requirements around goals and constraints of a specific system. We will look at those surrounding FlexGo's server system and choose that for analysis.

Goals of the Model

The final model should achieve the following goals:

1. Allow us to set expectations on servicing requests from various sources - specifically in terms of how many we can process within a "reasonable" amount of time.
2. Allow us to figure out where the bottlenecks in the system are, and where to scale to increase the overall capacity of the system.
3. Do all of the above in a cheap and efficient manner relative to brute force (deploying the full system, and actually doing empirical tests).

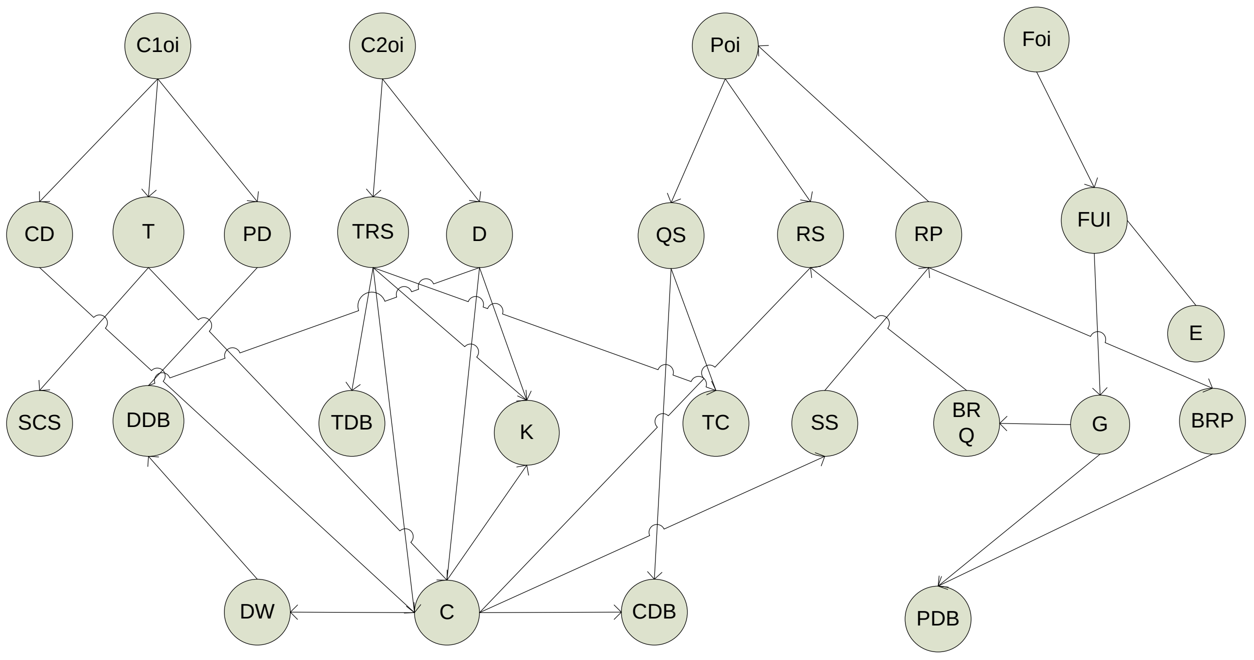
Constraints surrounding the Model

The model should satisfy the following constraints:

1. The system internally only understands messages. For the purpose of this model, it does not understand the sources. Each incoming request into the system is translated into a single message as it makes its way across the system.
2. Each service in the system has a specific service time for a specific message on a specific hardware. When the service is online, each service has a limit on the number of messages of a particular type it can process per second before the messages start queuing up and eventually "overflow" the service. This should not happen when all the sources are acting within their SLA's.
3. The service time for the same message may vary based on the service.

Constructing a Simple Model – Model 1

Let us start with the simple model where the services are nodes and the edges connect services that communicate with each other. A first cut at the real system today looks like the following:



Explanation of Model 1

1. This is a multi-commodity flow network.
2. The top row of nodes are all sources as well as sinks. These are the clients and partners.
3. The messages out of each source are all considered to be equivalent messages from a single commodity.
4. The arrowheads only indicate who initiates the communication. While asynchronous message-based communications are correctly represented, this means synchronous communications are also represented this way.
5. Each edge is assigned a capacity (per commodity) of total requests it can take, this can be calculated since the node (service) at the end of the directed edge has a certain capacity.
6. If we assume all the edges going out of the source have infinite capacity, then we can assess the max-flow using Network Flow algorithms to figure out what the SLA's are.
7. The min-cut will reveal the bottlenecks within with the flow network.

Limitations of Model 1

Here are some immediate observations on the above model:

1. The top row of nodes are all sources as well as sinks, there should be no other sources/sinks within the system. However, only some of them have arrowheads pointing back to them. This explains some difficulties in describing asynchronous messaging vs. Synchronous messaging. This is also true for several nodes within the system.
2. Several of the constraints are not taken into account, including:
    a. The translation between requests and messages is unclear. If they map 1-1, it violates the constraint that states that all messages are not necessary equivalent.

3. Capacity of the edges is defined per commodity. But the internal system cares only about the messages, not about the commodity. It is unclear how this capacity is measured and how to translate it to the internal system.

Improving on Model 1 — Model 2

Synchronous vs. Asynchronous Depiction

One of the primary limitations in Model 1 was the depiction of synchronous communications. They are currently represented as one-way directional edges. This introduces problems because this source is also the sink for that request. This however is solved by adding another edge with the same capacity from the target node back to the source.

Commodities, Messages and Capacities

Another primary limitation is the commodity-message relationship does not clearly aid in defining actual capacities. For example: If we have a client sending a message, and a partner sending a message of the same type, these messages are actually equivalent in the system. This is because the capacity for the service is not defined per commodity, but rather per message.

The solution to this is to define intermediate nodes that depict messages between the source and the system. This way, the capacity in the system is still accurately defined per message, and these intermediate nodes take care of the mapping.

One problem that hasn't yet been discussed is how to compute the capacities of each edge. In reality, these edge capacities are actually defined by the target node, so all edges ending at a target node have the same capacity. This capacity is simply computed by measuring the service time of a single message at that service. If at a particular service we know that a message takes a processing time of t, the capacity of the service is 1/t per second for that specific message.

Constructing Model 2

Instead of working on the real problem which is fairly complex, let us analyze the changes on an example problem that is representative of the primary problem.

Explanation of Model 2

1. There are 3 tiers. At the top is the source/sink tier. The middle is the message tier, and the bottom is the service tier.
2. For each service, let us normalize all the messages on a baseline Message 0. This way, all

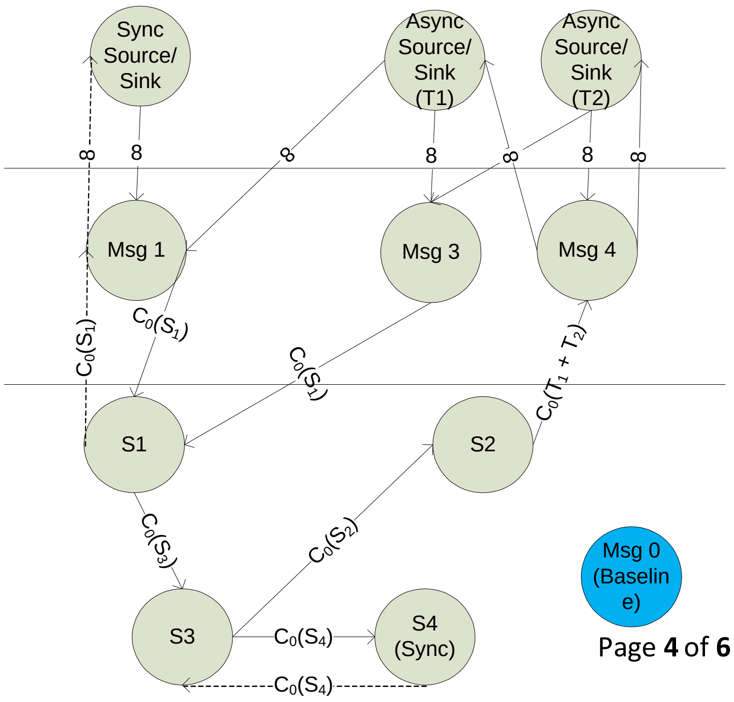


Figure 4

messages are normalized down to a single baseline message's processing time, and each service's capacity is represented as the capacity for processing the baseline message. If message 1 takes a processing time of t_1, we normalize such that the capacity of node S1 for message 1 is actually is $C_0(S1)$ which is nothing but $1/t_0$. Hence the capacity of the edge connecting the message 1 to the service node is $t_1/t_0 * C_0(S_1)$. This addresses the limitation of having the model represent everything in terms of message capacities, regardless of the source.

3. As a direct result of the above, this is now a single-commodity flow network. the commodity is message 0.
4. Also, as discussed earlier, synchronous calls are represented with an additional backward edge between the target and the source with the same capacity. This ensures that the source/sink relationship for that source is presented accurately.
5. Similar to Model 1, the max flow represents the SLA that can be guaranteed per partner/client.
6. Also similar to Model 1, the min-cut represents the first bottleneck in the system. In fact, anywhere that the flow = capacity is a bottleneck.

Open Challenges not addressed by Model 2

1. There is an assumption in normalizing that when 2 two dissimilar messages m_1 & m_2 are processed with each service time t_1 and t_2, the final time when both are done is $t_1 + t_2$. Sometimes, processing a single message may have some adverse effects on another message (like taking a global lock). This doesn't fit into the above model.
2. There are current implementations in the real system that split up the messages in a single service. This means a single message from the source can no longer be tracked as a single message. It's not clear how to fit this into the above model.

Complexity

Calculating the network flow once the model has been setup is straightforward by just application of the Ford-Faulkerson algorithm. However, setting up the problem involves additional layers of complexity including actually measuring the service time for each message time each service- O(nC) where n is the number of services and C is the number of messages. In the case of the current system in question, it's practically $O(n^2)$.

Applications

While the key applications are as suggested by the goals- SLA's and determining where to scale, this is primarily an offline algorithm. Additional applications exist if this is made online, such as a single backup machine can serve two different services based on the current computation of the bottleneck.

Alternate Approaches

Most solutions to this problem today are determined empirically- the existing infrastructure is tested for a certain load, and additional resources are added to address the current pressing problem. In practice, this works well most of the time. But the relative demerits when compared with the solution in this paper are obvious: the costs associated with testing a full scale-out, the surprises involved when actually hitting the limits etc. The demerits of the current solution include having to characterize each service's processing time per message, and each of the open challenges listed above (dependent on good programming practices, also works only with certain implementations).

Concluding Remarks

The fundamental problem is analyzing the capacity of a complicated distributed system. Testing the real distributed system for various outcomes can be a complex process. Instead, utilizing network flow, it is fairly simple to characterize each service and construct the full model for the system.