Flock: A Reliable Method for Dynamic Resource Federation

Team TBD: Drew Sorrels, Hassan Almas, Andriy Katkov, Dustin Pho, Edward McEnrue
Faculty Advisor: Dr. Ali Butt

1. Abstract

Flock enables efficient resource discovery and organization on ad-hoc wireless local area networks (WLAN). It does so by combining gossip, consensus, and container based protocols to provide fault tolerant allocation of Docker-based images. This concept has been widely implemented for data center networks, but not for networks that are ad-hoc. A ad-hoc WLAN solution provides a means to federate off the shelf machines at public gatherings, as opposed to using a more expensive data center solution. To further improve the capability of the system, we introduce a dynamic alpha function for peers to utilize for organizing resources in a manner that suits the particular network.

2. Introduction, Background, and Enabling Technologies

Over the past decade, there has been a myriad of infrastructure tools that provide resource federation functionality for data centers [1][2]. Data center services like Amazon Web Services, Google Cloud Platform, and Microsoft's Azure all provide resources en masse, but they can also be very expensive. Therefore, a tool that federates resources on WLAN that can resist the detrimental effects of ad hoc connections provides value to users who have machines, but no funds to allocate for a data center service. However, this ad hoc WLAN solution would also need to compete in terms of usability, that is, provide generic resources to allow for quick and painless deployment of applications. Flock was built in order to accomplish these goals.

In particular, Flock aims to answer the following questions for an ad-hoc WLAN: what machines are currently on the network; what do the other machines know; what should the machines do; which machines will do it? All of these questions must be answered within the context of fault tolerance and efficiency to provide a decentralized service that is just as stable and performant as data center centralized services. Furthermore, the answer must provide a generic solution, so that application deployments do not have to be tailored for the peer machine's architecture.

These questions are answered by using gossip, consensus, and container-based protocols. In order to discover what machines are on the network, Flock uses a standard broadcast over the router and has each node of the cluster listen for these. If any new node joins, this is effectively the mode of entry. The discovery subsystem hooks into the gossip subsystem to spread the arrival or exit information of a node. In order to figure out what other machines know, Flock uses a publish-subscribe pattern through a Java server implementation to respond to a node asking what Docker images are currently available for use on the cluster. Similarly, this server provides a way to stop Docker containers or provision them over the system. Finally, a preference-based election is implemented with the bully algorithm to provide an answer for which machine will execute leader instructions for the cluster. Section 3 provides further details on the specifics for each of these answers. In order to have Flock provide a generic solution, Docker Machine's API is used to create custom Dockerfiles based on Flock's API server.

The approaches Flock uses have been used in other widely used distributed systems [3][4]. Flock integrates these solutions and protocols because they provide the most decentralized but fault tolerant solutions for each subsystem. Likewise, in order to implement preference based elections, system summary information is parsed and linearly combined based on a weighting scheme provided by the user.

3. Design and Implementation

Flock's design from its inception was always meant to be modular, such that additional subsystems could be added easily. In this way it mimics data center services like AWS or Azure. Each component of Flock queries other components to receive the necessary information to proceed.

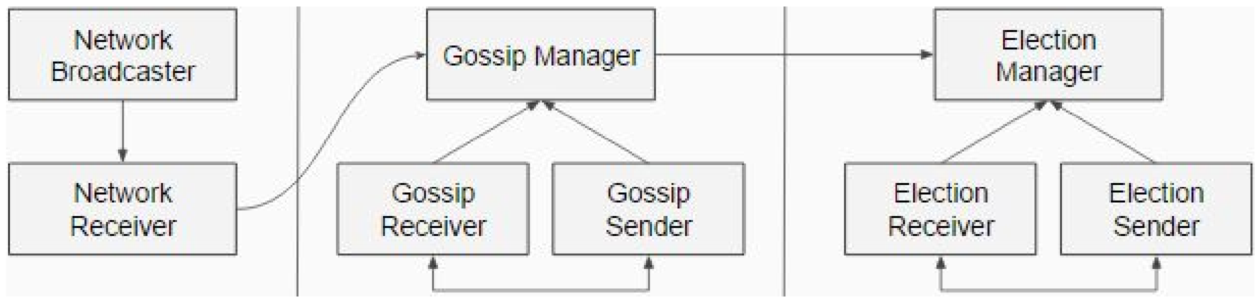

Figure 1: The system design and flow of communication between the network discovery, gossip, and election layers

In Figure 1, we see the election manager acquiring a node list in order to initiate an election on a complete topology. The three core services, network discovery, gossiping, and election, provide information for node entry, cluster membership, and leader selection respectively.

3.1 Gossip

Cluster membership is handled by the Gossip Manager which uses standard UDP sockets to communicate with other nodes. The Gossip implementation is based off a variation of the initial gossip protocol in which the manager randomly selects a node to communicate membership details with each round [5].

Each node that runs gossip will keep some meta-information about other nodes that it communicates with. This includes a UUID, an IP address, a heartbeat, a generation time, and a status. At every timestep of gossip protocol, we send a list of all node metadata stored on this node to another random node. The other node merges the received list of nodes with its list of node metadata with the most up to date information. It then sends back a list of all nodes that were different and newer, or not present in the received list. That list is then sent to the original sender which merges the list with its own node list.

Gossip handles ungraceful exits by periodically checking the amount of time since it last heard from all nodes. We set a timeout of 30 seconds. If we have not heard from a node in that time, then the node is marked as dead and removed from the list. This process is undertaken by all nodes in the cluster regularly, so each node will check and remove any nodes that it has not heard from recently.

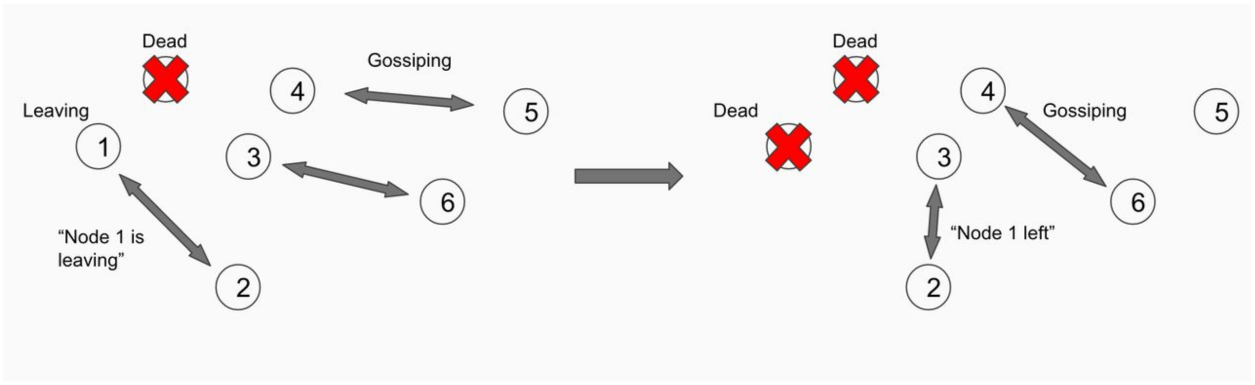

Figure 2: An iteration of our gossip protocol in which a node is leaving and another node dies. Node 1 notifies its gossip partner (node 2) that it is leaving. In the next iteration of gossip, node 2 notifies a partner that node 1 left. These two nodes will then notify their partners in the next iteration that node 1 left.

Graceful exits are handled by marking the current node as leaving, then undergoing one last gossip round to allow this information to propagate through the network. The node that receives this information will update its node list (Figure 2). Other nodes can then gossip and receive these updates. Ten seconds after the heartbeat time, the node is marked as dead and removed from the list along with other nodes that have not been heard from recently.

3.2 Election

The election manager also handles its execution using standard UDP sockets for communication. The algorithm for electing a leader is derived from the bully consensus algorithm [6]. The bully algorithm is a preference-based leader election algorithm. For Flock, the preference function, alpha, is determined by a linear combination of features which represent system properties like, Flock uptime, load average, IOWait, and latency to a common DNS. Each feature of the function is weighted on start of an election to select the most desirable properties for that given user.

Elections are executed synchronously, gracefully handling network churn (including loss of leader) and network partitioning. The election will always elect the leader with the highest alpha in the cluster.

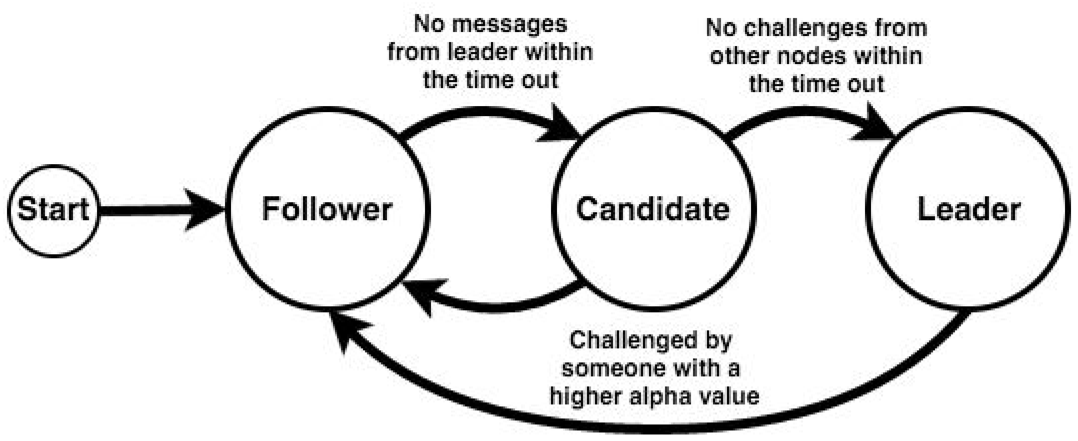
No messages from leader within the time out
No challenges from other nodes within the time out
Start
Follower
Candidate
Leader
Challenged by someone with a higher alpha value

Figure 3: FSM that describes the bully leader election state changes.

Our implementation of the Bully Election algorithm included three node states - Follower, Candidate, and Leader (Figure 3).

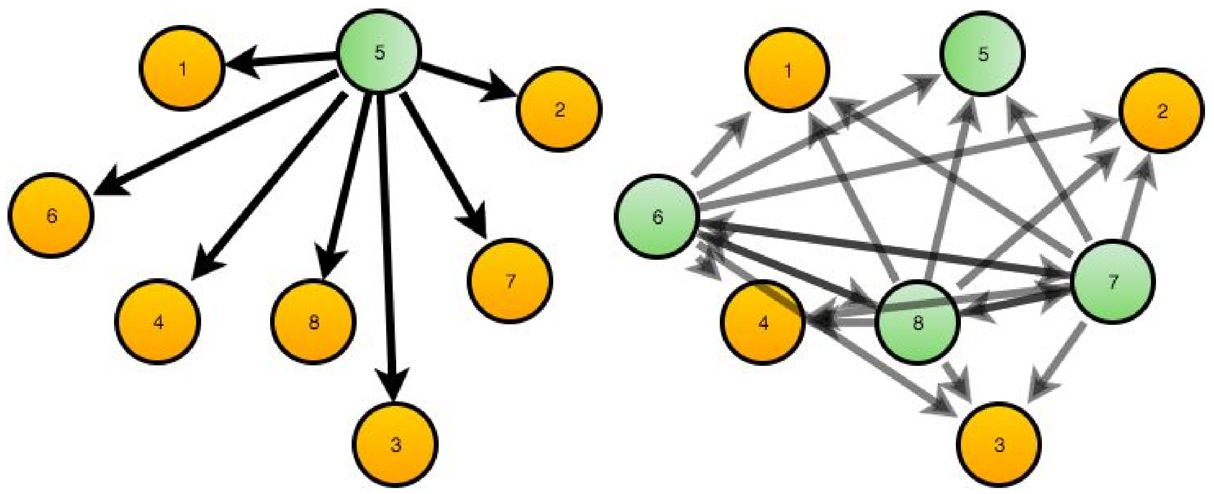

Figure 4: An election is started by the node with an alpha value of 5.The node that originally started the election waits for responses from the cluster. Nodes that received the election request transition into candidates and join in if their alpha values are better.

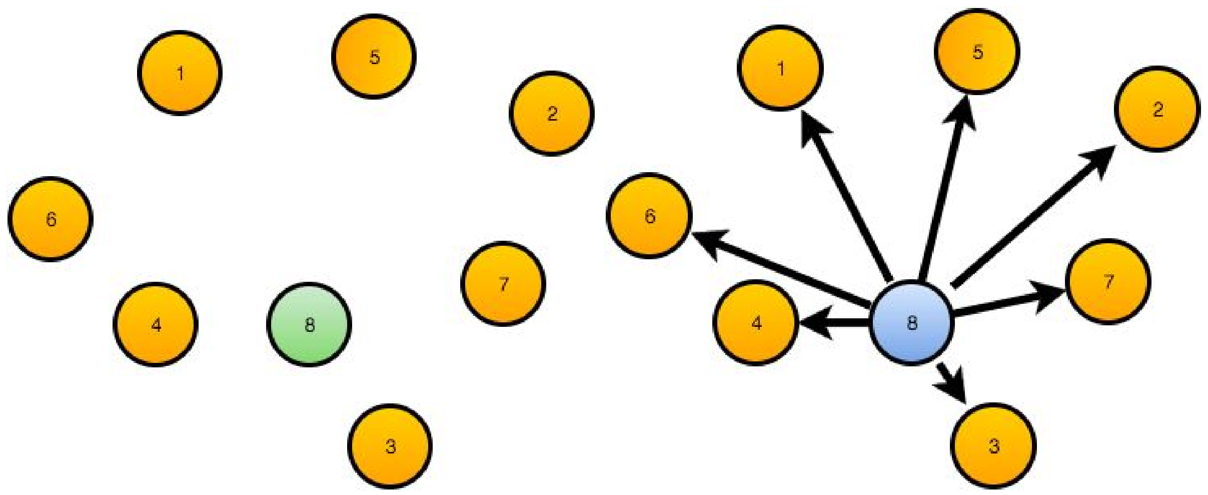

Figure 5: All candidate nodes step down once they see that a node has an alpha value of 8. The leftover candidate node waits for a response. The candidate node remains unchallenged and promotes itself to leader. It begins suppressing the cluster.

A Follower transitions into Candidate state only when it receives an election message by a Candidate with a lower alpha value. It then sends a response to the Candidate telling it to stand down. It then broadcasts out its own election message and waits (Figure 4). A Candidate can assume the position of Leader when there are no challenges to its election message (Figure 5). A Leader maintains control over the cluster by sending out periodically sending out suppression messages. If a node in Follower state does not receive messages from a Leader, it will transition to Candidate state and start an election on the cluster. Nodes will step down if an election is started by a Candidate who has a higher alpha value, regardless of its current state. Suppression and election messages are broadcasted out to the entire cluster. Responses to election messages (telling the other Candidate to stand down) are directly sent.

3.3 Flock Server

The core Flock services provide the necessary solutions for resource discovery and organization, but they do not handle creation of those resources. Instead, an API layer is built on top of Flock to handle communication with the user of the service via a terminal based front end.

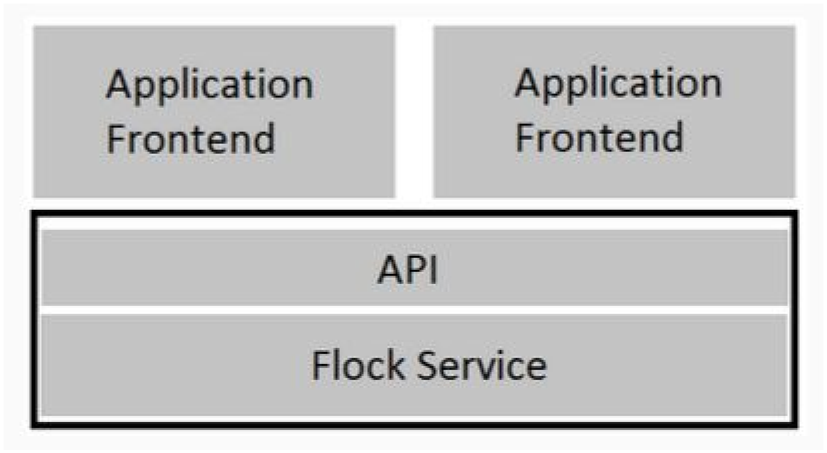

Figure 6: The system design with both the Flock service and user's front ends

The API was designed as a TCP server to allow external applications to receive information about the cluster. The server itself is multithreaded to allow multiple application frontends to connect and gain information about the cluster (Figure 6). This allows Flock to run as a separate service that provides information to multiple applications that will run on top of it.

The API takes in JSON objects as a request for information. The "type" field determines what information is desired. Depending on the type, some additional information may be required as well. For example, if creating a "has_image" request, you must also provide the image that you want to test. These requests are then processed and some JSON output is sent back to the client.

For requests that require communication with other nodes, we run a separate multithreaded TCP server. This other server runs on a separate port and only processes certain commands. This design is used to simplify additions to the cluster. Internode requests go in one server, intranode requests are processed by a separate server. Currently, the only internode requests that we support are "has_image" and "run_image". These commands return true/false depending on whether the image exists and was started. From here, the API server that requested this information generates a list of nodes that started the image.

Another feature of our server is the ability to observe node enter/left events as well as leader changed events. This is accomplished via Java Observer/Observable. When our gossip protocol removes nodes or receives a new node, it notifies observers with a JSON object. This contains some information about what type of event it was as well as the information for that event. This allows the API to send changes in cluster membership and leader changes.

In terms of scalability, the API does not pose any issues. There is a linear increase in messages sent between nodes for certain commands. Any single node command is just a query to our gossip manager or election manager, so those are very efficient regardless of the size of the cluster as they do not have to communicate with other nodes to complete the queries.

3.4 Docker-Machine Integration

The final component of Flock is its service for allocating Docker based containers on each peer node. The Docker ecosystem is accessed through a custom shell script which interacts with Docker, Docker Machine, and VirtualBox. The general flow for starting up a container through this script is like this: for non-Linux OSes, the first step is to open ports on the Linux VM which will be running Docker images. This is required because when we expose a custom port for SSH into a new container in the Dockerfile, this will only expose it to the VM. Then, we need to forward this port from the Linux VM to the host OS (OS X or Windows) so that the port can be reached from the host OS's IP address.

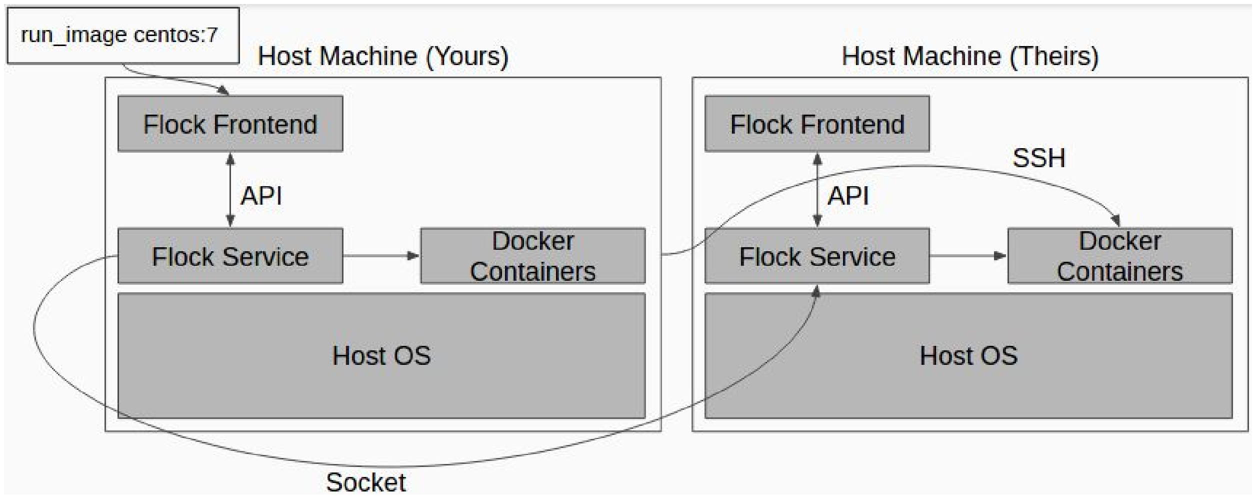

Figure 7: The full system design with docker integration.

Once we have set up the ports appropriately for SSH access for a future container, we can start a container. Then, "flock_docker.sh start [DOCKER_IMAGE] [PORT]" will start the given Docker image with SSH available on the provided port. The way the start command works is that it will take the passed Docker image and insert it into a custom Dockerfile as the new base image. This custom Dockerfile contains setup instructions for the SSH server that will be used for communication with this container. Then, this new image is built using docker build and then started with the given port exposed on the container. The image can also be stopped using “flock_docker.sh stop [DOCKER_IMAGE] [PORT]”. This script is bundled with Flock and is run on the nodes in the cluster when the API invokes it locally (Figure 7).

4. Evaluation

Flock was constructed mostly as a prototype to answer the question of “can an infrastructure tool be made for ad hoc WLANs?”. Overall, Flock does successfully demonstrate that it is possible. Each component of Flock scales well and provides fault tolerance. However, there are still many flaws in regard to security, some of which are brought forth in this section.

4.1 Scalability

In terms of scalability, gossip has a theoretical completion time of O(log(n)) rounds [7]. Adding nodes to the cluster is done with minimal impact on the resources needed. By adding a single node to the list of nodes, an additional 2 messages are being sent every round (one from the node to a partner and one back to that node). The lists of node metadata contains one additional node, which in a JSON representation takes less than 128 bytes. This could very easily be shrunk down by using a datatype other than JSON to transmit node information.

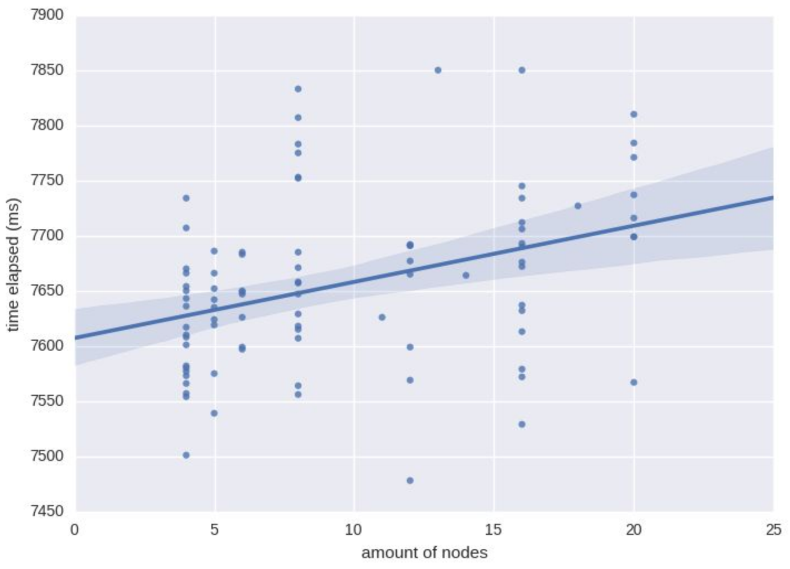

Figure 8: FSM that describes the bully leader election state changes.

For elections, as the size of the cluster increases, the time it takes to reach a consensus on the leader also increases at a linear rate (Figure 8). Since Flock is generally used on smaller clusters, this does not become a problem. However if we were to apply our election algorithm to a much larger cluster, the inefficiencies would become apparent.

The outer components, the API along with its container allocations, scale linearly with the requests that come to the system. If the ports are properly opened for SSH access, and there is no port collision between various containers, then the scalability of having multiple Flock jobs running multiple containers is very feasible.

4.2 Security

There are many improvements that can be made for Flock in order to reduce the amount of threats. In particular, its algorithms do not account for malicious cases. Similarly, the API was designed for benevolent users. Therefore, in order for this to be used outside of a casual environment, in depth penetration testing would need to be made. However there are a few security problems that currently only require minor modifications in order to be fixed.

For example, the gossip implementation will allow malicious nodes to enter the cluster. There are no protections against a node sending false data. For example, a node could enter the cluster and modify the data of all nodes in its list. This could be done in such a way (setting generation time to some point in the future and setting heartbeat to a time in the past more than 30 seconds ago) that would cause other nodes to remove nodes from their list. However, unless it did this all at once, the cluster would recover by other nodes gossiping to the node that had this malicious data sent to it. Furthermore, we periodically send network discovery broadcasts, which will allow us to reconstruct the cluster by rediscovering nodes and adding them to the node list.

In a similar case, for leader election, a malicious node that participates can spoof its alpha value in order to get itself elected leader. Depending on the service being run, privilege escalation could be a serious concern. If the cluster is running Spark, the malicious node would then have the ssh-keys to every other node's Spark container. For both gossip and leader election, byzantizing the algorithms or choosing a byzantized algorithm would be necessary.

One of the flaws in the API is a lack of security features. We do not enforce any sort of authentication or denial of requests. Any malicious user (once finding the API port) may send as many requests as they want. They could start thousands of containers on the cluster without any way to stop them. This is obviously an important issue, as one malicious user could prevent a cluster from accomplishing any work.

Finally, we assume Docker is secure in its implementation of containers, but other security flaws do exist. For example, there is no job identification or limiting authentication that prevents a node from starting a large number of jobs to strangle resources on other nodes. Any resources distributed to another node's container will remain there if a node leaves the network.

5. Conclusion and Future Work

Flock automates many of the typical resource federation processes for ad hoc WLANs. It provides fault tolerance and scalability through a loosely coupled system design of well tested distributed algorithms and resource allocation libraries. Furthermore, Flock is a completely free alternative to data center infrastructure in ad hoc environments where multiple commodity machines are connected. Regardless, there are still many problems to be solved to make Flock be production ready.

At the consensus level, there is of course the possibility that even the node with the best alpha will fail contrary to its alpha features predicting that it wouldn't. In this scenario, what would be the best approach to take? Some leadership election algorithms trade the ability to have multiple failover nodes for a higher initial runtime cost [8]. This tradeoff would be ideal, since it’s unlikely that the election in combination with alpha predictions would fail.

There are also many possible features at the container allocation level. A node can connect to another node’s container by using the other node’s IP and the specific port for SSH used for the job. The SSH server is added on top of the given image for the job to allow access. This worked for some of the simpler images such as centos or ubuntu; however, it presents problems for more complex images, especially ones that have their own SSH setup such as Spark. An improvement to this in the future would be to embrace Docker’s one process per container philosophy and have a compositional SSH server. This would reduce any collisions between the desired Docker images for a job and any SSH behavior inside of them. It would also allow us to provide exactly the Docker image that they request for a job without the modification of adding the SSH server. We attempted to use this compositional approach, but we’re unable to make it work with our system in the time we had.

With these additional features, along with addressing the security concerns mentioned previously, it is possible that Flock would become a system that is useful even at the industry data center level. However, the question of whether an infrastructure tool can federate ad hoc WLAN resources is fully answered by Flock.

Appendix

1. Hindman, Benjamin, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." *NSDI*. Vol. 11. 2011.
2. Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.
3. Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review* 44.2 (2010): 35-40.
4. Butt, Ali Raza, Rongmei Zhang, and Y. Charlie Hu. "A self-organizing flock of condors." *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003.
5. Van Renesse, Robbert, Yaron Minsky, and Mark Hayden. "A gossip-style failure detection service." *Middleware'98*. Springer London, 1998.
6. Garcia-Molina, Hector. "Elections in a distributed computing system."*Computers, IEEE Transactions on* 100.1 (1982): 48-59.
7. Wuhib, Fetahi, Mads Dam, and Rolf Stadler. "A gossiping protocol for detecting global threshold crossings." *Network and Service Management, IEEE Transactions on* 7.1 (2010): 42-57.
8. Singh, Awadhesh Kumar, and Shantanu Sharma. "Elite leader finding algorithm for MANETs." *Parallel and Distributed Computing (ISPDC), 2011 10th International Symposium on*. IEEE, 2011.