Dashaylan Naidoo, Edward Huang, Kamil Wasty, Vincent Tan

HiveMind: Project Report

6th April 2018

ABSTRACT

HiveMind is a distributed shared memory (DSM) system built with Go and designed based off the TreadMarks system. Distributed shared memory is meant to simulate shared memory across multiple processes which may or may not be on the same physical machine, this eliminates the need for other communication protocols such as message passing. This paper will discuss the details of our design including the structure and workflows of our project, how ShiViz and GoVector are used in our project and finally how our system performs.

OVERVIEW

Introduction

A software *distributed shared-memory* (DSM) system allows a network of workstations executing processes in parallel to share memory, even if the workstations themselves physically do not. This is an attractive approach to parallel computation as it allows programmers to forgo the traditional message passing paradigm, which can be considered more difficult due to the need for explicitly partitioning data and managing interprocess communication. Instead, they are able to focus on algorithmic development thanks to a global address space and leave the unruly parts to the underlying DSM to handle [**1,4**].

TreadMarks is a DSM created by Pete Kehler et al. to run on Unix systems utilizing the C programming language. It featured unique release consistency and multiple writer protocols, utilized many features of the Unix system such as depending on the ability to modify the protection status of virtual memory pages and also add custom handlers to trap on access violations to the shared memory. The handler was able to perform actions to support the TreadMarks core functions. At the time of its, TreadMarks release was considered state of the art.

HiveMind is an implementation of the TreadMarks DSM as a Golang package to run on Linux workstations. It features the core components of TreadMarks, including barriers, lazy release consistency, lazy diff creation and multiple writer protocols.

An immediate challenge that became apparent when using the Go language was the lack of native support for installing custom segment violation handlers. In order to overcome this limitation we implemented a package in Go that emulated the behavior of the shared memory model required by TreadMarks.

In our system every node or processor is called a drone. Each drone can read/write from the shared memory and act as a lock manager for a specific lock. A special drone, which is usually the first drone alive, acts as a centralized barrier manager.

DESIGN

Architecture

Figure 1 below shows the Go packages that make up the HiveMind implementation. The HiveMind system was built in a modular way. The core HiveMind capability is contained in the hivemind package. It implements the TreadMarks interface and the core Lazy Release Consistent (LRC) algorithms defined in TreadMarks. The Shm package implements the shared memory interface. The communications between the nodes in the HiveMind system is done through the IPC package. IPC package is responsible for creating the point-to-point connections between the nodes, handling the sending/receiving of opaque messages between the nodes, and handling any network level failures and recovery. The config package was added to support the configuration files that define the parameters needed to establish the connections to the nodes, such as IP address, SSH credentials, etc. The IPC also has a dependency on the golang.org/x/crypto/ssh package used to support ssh operations. The HiveMind package has a dependency on the github.com/arcaneiceman/GoVector/govec to generate a ShiViz-compatible vector-clock timestamped log of events for HiveMinds.

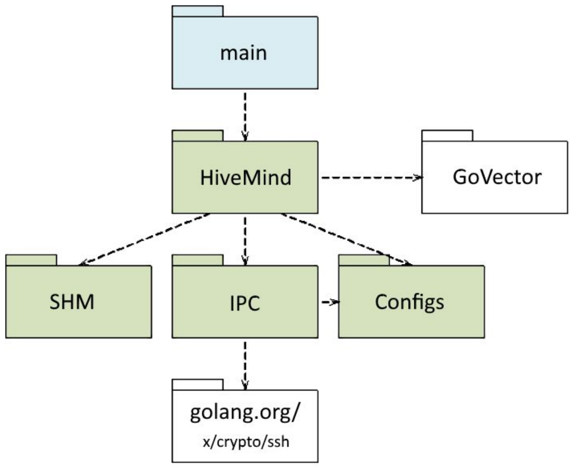


Figure 1: Overview of the HiveMind project structure

HiveMind Package

The HiveMind package implements the core logic to drive the TreadMarks compatible DSM system. The core data structures are modeled after the data structures described in [**1, 2, 3,4**] and consists of the data structures shown in Figure 2.

The HiveMind package maintains a single data structure that contains the following primary data structures:

- Page Array - with one entry for each shared page. Each entry in the page array contains:
  - The current state: no access, read-only access, or read-write access.
  - Copyset - An approximate copyset specifying the set of processors that are believed to currently cache this page.
  - Twin – snapshot of the page taken before any changes are made to the page.

- ○ Write Notices - For each page, a two dimensional array with each row representing a processor and contains an array of write notice records corresponding to write notices received from that processor for this page. If the diff corresponding to the write notice has been received, then a pointer to this diff is present in the write notice records. This list is maintained in order to decrease interval indices.

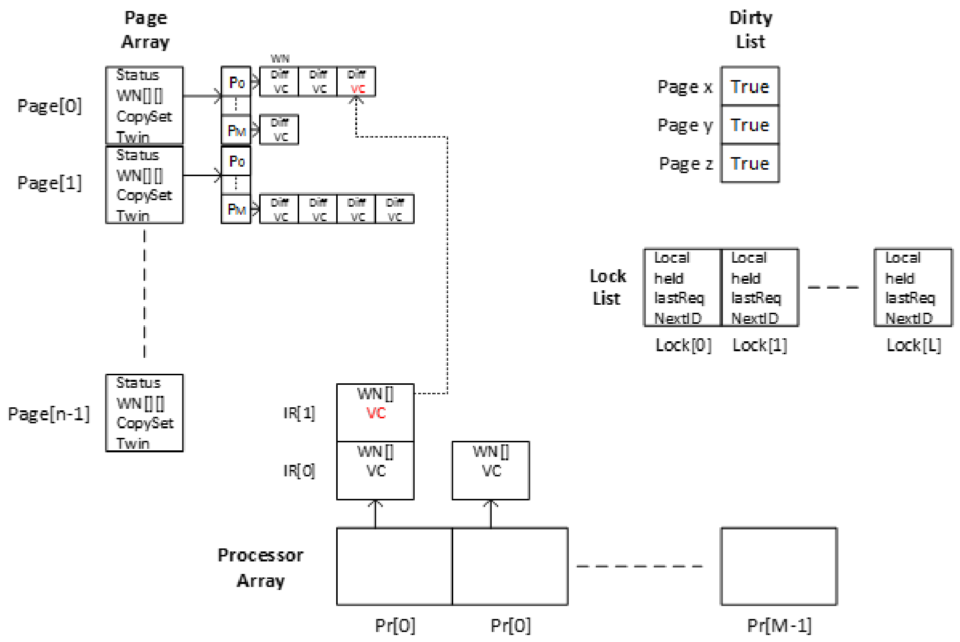


Figure 2: Primary data structures of HiveMind

- Proc array - with one entry for each processor. Each entry in proc array contains a pointer to array of interval records, representing the intervals of the processor that the local processor knows about. This list is also maintained in order to decrease interval indices. Each of these interval records contains a pointer to a list of write notice records for that interval.
- Interval records - containing mainly the vector timestamp for that interval. Each time a process executes a release or an acquire of the shared memory, a new interval begins and the interval index is incremented. Intervals of different processes are partially ordered by the vector timestamp.
- Vector timestamp - contains an entry for each processor. The entry for processor p in the vector timestamp of interval i of processor p is equal to i. The entry for processor q != p denotes that most recent interval of processor q that precedes the current interval of processor p.
- A set of write notice records. Each write notice is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. The actual modifications are carried by diff.
- A diff pool. A diff is a run length encoded record of the modifications to the page, created when processor requests the modifications to that page or a write notice from another processor arrives for that page.
- Dirty List – map of pages that were modified in the current interval. This map is cleared when all of the changes are incorporated into the write notices.
- Lock List – array of Lock objects to keep track of the HiveMind locks. The local field is set true when the current processor has the lock token, the held field is true when the lock is held by

the processor. The lastReq field is set to the last processor requesting the lock and the nextID points to the next node to send the lock response when the lock is released.

Shared Memory Module (shm)

The main data structures that make up the shm package is shown in Figure 3. There is a single structure which contains the following components:

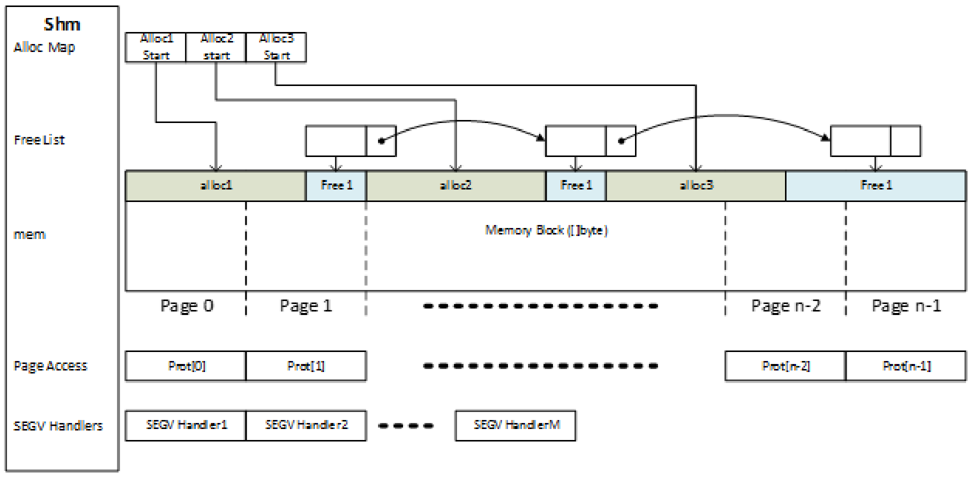


Figure 3: Overview of the shared memory module

- Memory block (mem) – this is an array of bytes. The size is equal to the number of pages * page size. All of the shared memory offered by the HiveMind API resides in this block of memory.
- Free List – is a linked list of free chunks of memory blocks. Each node in the list points to a contiguous block of free memory. At startup, there is one free chunk. When a previous allocated chunk is returned to the free list, it is coalesced with adjacent nodes. The free list always maintains the minimum number of free chunks.
- Alloc Map – keeps track of the allocated blocks of memory. Every call to malloc() will locate a free block of memory of the specified size in the list of free chunks. The size of the free chunk is reduced by the size of the allocated block. The address of the allocated block is used as the key to the Alloc Map and the value is set the size of the block.
- Page Array – maintains the access status of each page of the memory block. The page can be set to No Access, Read, and/or Write.
- SEGV Handler list – maintains a list of segment violation fault handlers. These handlers are installed by the user. The handlers are called when a access (read or write) occurs to page and the current access to the page is violated, for example if the access rights of the page is No Access and a user attempts to read or write to a block of memory that spans the page, then the fault handler is called.

Configurations JSON

HiveMind stores information about the network in a json file. HiveMind uses this file to determine whether it is the CBM or where to connect to CBM. The file also stores the addresses and PID of all other initial drones.

IPC Module

HiveMind drones communicate over TCP connections. The IPC module provides channels to which the memory module can place and receive messages from. The IPC module handles connecting to other modules, reliable and low-latency connections between drones, and deployment of the HiveMind library and the app using it onto other machines, given in a configuration file.

We chose to minimize IPC and memory module interdependence by utilising a pair of channels (rx channel and tx channel) where arbitrary messages can passed utilizing them. This isolates the implementation of one module from the other, allowing both to be developed rapidly. This also allows refactoring one module without having to refactor the other module too. Only messages sent into the tx channel will be forwarded to the target drone by the single sender in IPC module. And on the other hand, messages will be received by (multiple) receive handler(s), and only messages sent into the rx channel will be visible to the HiveMind Module.

HiveMind API

API	**Description**
Startup()	Starts the HiveMind DSM on all processors. It should be the first call after creating the HiveMind instance.
Malloc(size int) (int, error)	Allocates a block of free memory and returns the start of that block
Free(addr int) error	Frees a previously allocated block of memory.
Barrier(b uint8)	Barrier blocks the calling process until all other processes arrive at the barrier.
LockAcquire(lock uint8)	LockAcquire blocks the calling process until it acquires the specified lock
LockRelease(lock uint8)	LockRelease releases the specified lock
GetNProcs() uint8	GetNProcs gets the number of processors configured for this DSM
GetProcID() uint8	GetProcID gets the processor ID of this process
Read(addr int) (byte, error)	Read a single byte at the specified address
ReadN(addr int, len int) ([]byte, error)	Read an array of bytes starting at the address
ReadFloat(addr int) (float64, error)	Read a single float64 starting at the address
Write(addr int, val byte) error	Write a byte at address
WriteN(addr int, data []byte) error	Write an array of bytes starting at the address
WriteFloat(addr, val float64) error	Write a single float64 starting at the address
Exit()	Exit is called to shut down the HiveMind DSM.

System Flow

Figure 4 shows the tasking model and information flow in a typical HiveMind application instance. The application interacts with the HiveMind API in the context of the main go routine. A background receive go routine is created when the StartUp() API is invoked. Both the main and receive tasks access the shared memory and global HiveMind data structures. Each of the shared data structures has a mutex to control access to the structure. As mentioned in the IPC section, the TxChan and RxChan channels are used to pass messages between modules. HiveMind also has a third channel which is used to block the main task until a response message is received.

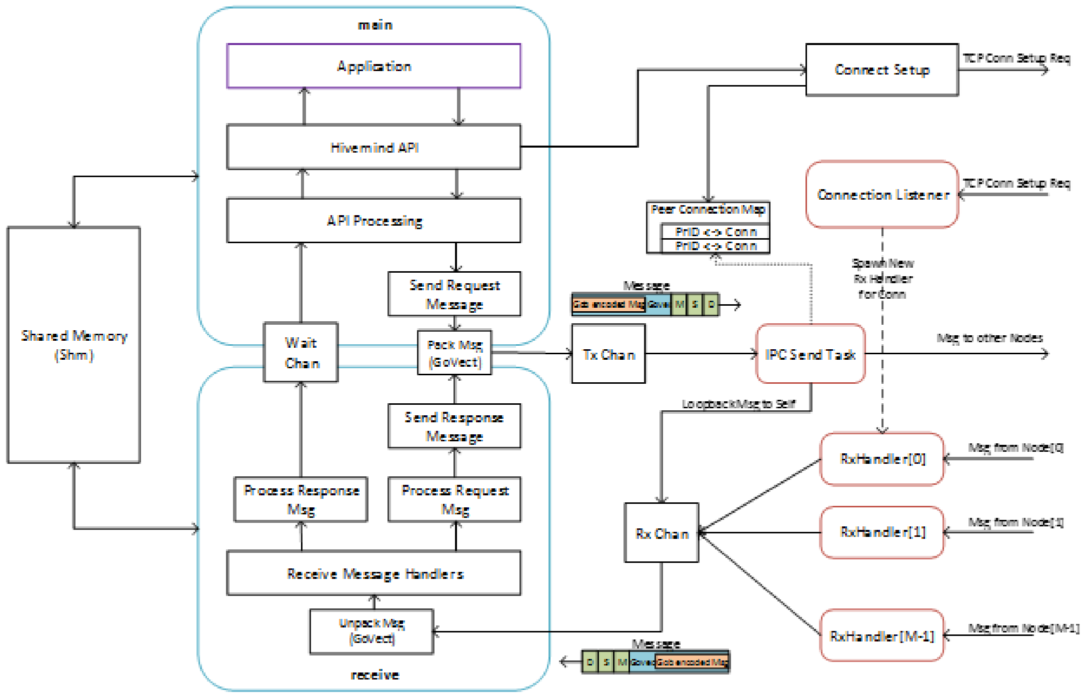


Figure 4: State diagram for a drone in HiveMind

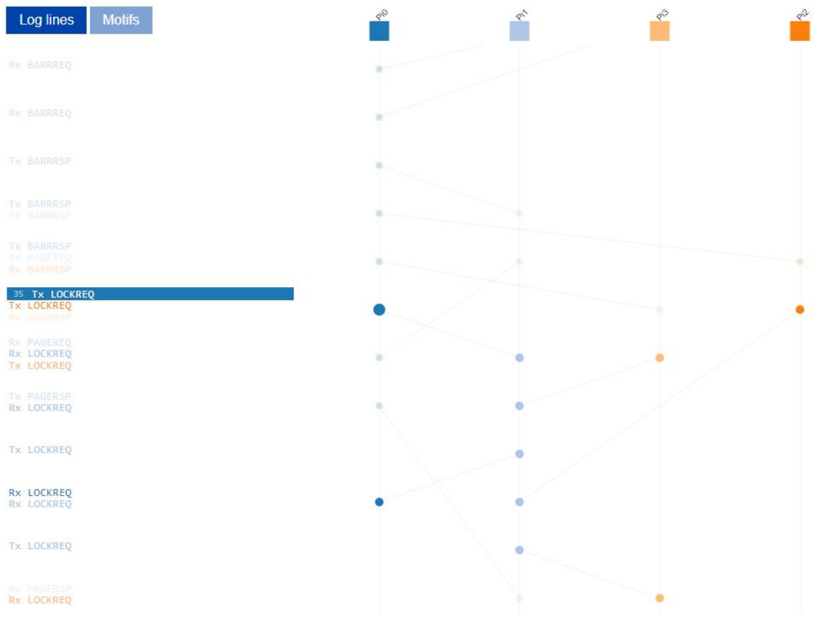
The message flow will be clearer if we walk through an example. We consider the sequence when the user invokes the LockAcquire() API. When the LockAcquire() call is invoked, the HiveMind processing layer will determine that it needs to send a LOCKREQ message to the Lock Manager for that lock. It encodes the message and calls the Send Request Message with the destination process id (D) of the Lock Manager. After the Send Request Message function returns, the LockAcquire() then blocks until it gets input from the WaitChan. The message is the encoded using the gob encoding and handed to the GoVector PrepareSend() to log the event in GoVector and encode it. The GoVector payload is then prepended with a three byte header: D=destination ID, S=this processor ID (i.e. Source), M=message ID.

The message is written to the TxChan. The message is picked up by the IPC send task which looks at the D byte in the header. If D is not equal to the source then it is prepended with a uint64 containing the length of the message and sent out on the connection registered in the map table for D. If D is the same as the source then it just puts the message back on the RxChan. At some point later the LOCKRSP message is sent back to this node. It arrives at one of the IPC RxHandler tasks that handles messages from destination ID of the original request. The uint64 length on the message is striped off the message and the message is put on the RxChan. The Receive task in HiveMind removes the message from the channel and uses the 3 byte headed to identify the source ID (S) of the message and the message ID (M). The GoVector

UnpackReceive() is called to unpack and log the event. The message is then decoded using gob to get back the message in Go data structure. The response handler processes the message and puts a message on the WaitChan to unblock the application.

GoVector and ShiViz Integration

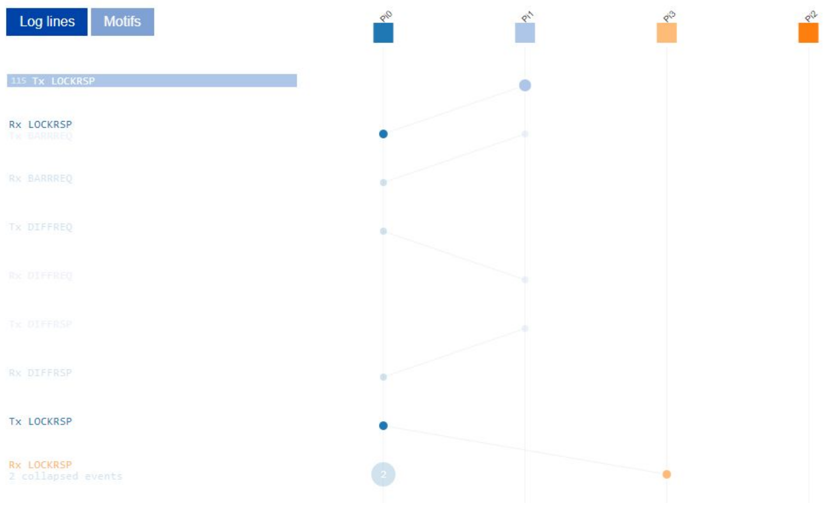
As mentioned earlier, the GoVector package is integrated with the HiveMind messaging to generate logs for ShiViz, enabling users to visualize how drones interact with one another.



```
Rx BARRREQ

Rx BARRREQ

Tx BARRRSP

Tx BARRRSP
Rx BARRRSP

Tx BARRRSP
Tx PAGEREQ
Rx BARRRSP

35 Tx LOCKREQ
Tx LOCKREQ
Rx BARRRSP

Rx PAGEREQ
Rx LOCKREQ
Tx LOCKREQ

Tx PAGERSP
Rx LOCKREQ

Tx LOCKREQ

Rx LOCKREQ
Rx LOCKREQ

Tx LOCKREQ

Rx PAGERSP
Rx LOCKREQ
```

Figure 5: Lock Requests & Manager Redirection

These ShiViz figures highlight lock acquisition and release, along with the end of a barrier at start of Figure 5. In this code, Pi1 is managing lock 1 and starting at the highlighted event in Figure 5, all other nodes send a lock request message to Pi1. As a lock manager, Pi1 remembers who the last requester of the lock was and forwards subsequent lock requests, forming a chain of waiting nodes. In Figure 6 we see Pi0 acquiring lock 1 from Pi1 at the top of the diagram. Once finished, instead of returning it back to Pi1, it sends the lock to Pi3 who had requested the lock from Pi1 shortly after Pi0 did. The use of managers greatly reduces the amount of messages needed by the DSM.



```
115 Tx LOCKRSP

Rx LOCKRSP
Tx BARRREQ

Rx BARRREQ

Tx DIFFREQ

Rx DIFFREQ

Tx DIFFRSP

Rx DIFFRSP

Tx LOCKRSP

Rx LOCKRSP
2 collapsed events
```

Figure 6: Lock Release

IMPLEMENTATION

Drone Deployment

The IP addresses of each Azure virtual machine, and their credentials, are stored in a configuration file. During the start up of the first drone, it reads this file and then creates a drone configuration file that contains IP addresses of VMs, the assigned PID, and the IP address of the CBM. Then, an executable is compiled and copied along with the drone configuration file onto each drone. Finally, a shell script is invoked which SSHs into each drone and starts up the application in the background.

EVALUATION

Methodology

To test the system, we wrote several different applications utilizing shared memory, and compared them to running said applications normally. For evaluation, we ran the Pi.go app, which calculates the value of Pi, and we measure the time taken and the error compared to a hard-coded constant value of Pi. We ran all tests with 10 million iterations.

Results

To evaluate our design and system, we ran a demo program that performs Pi approximation using integration. As a hypothesis, we believed that the time needed to complete the program will be improved, that is, decreased, as the number of drones participating in the calculation increases.

First of all, the program and the system is running in the Azure environment with limited resources. This cause some performance fluctuations and disruptions to the system. After running 250 times with different setups, in 4% of the times the program will not terminate.

Other than varying the number of drones used to run the program, we only vary one variable in order to evaluate the system. As an important factor that controls the complexity of the program, there is a variable, n, that resembles the number of iterations to be performed in the program by each drone. We selected 3 levels of n: 1000000, 50000000, and 100000000.

With n = 1000000, despite that there is no speed up observed, there is even a speed down. This is caused by the overhead in our system, in which examples would be message passing between drones, and/or network latency.

n = 1000000		
Number of Drones	**Time Taken (ms)**	**Speedup**
1 (Serial)	17.0429493	1x
2	31.9371703	0.5336399293x
4	60.778193	0.2804122409x

6	79.5097129	0.2143505325x
8	104.8438521	0.1150521085x

With n = 50000000, the improvements that HiveMind brings to the program start to become visible. Most notably, in this setup, running the program with 6 or 8 drones may speed up the calculations up to 2.2 times.

n = 50000000		
Number of Drones	**Time Taken (ms)**	**Speedup**
1 (Serial)	602.4116781	1x
2	666.6268153	0.903671536x
4	399.7302206	1.50704562x
6	286.8873091	2.099819891x
8	275.5670257	2.186080416x

N = 100000000 is the largest we can use with our system in the Azure environment. In this set up, running the program with 1 drone (that is, serial) takes 1.2 seconds on average. And with 8 drones in the HiveMind system, it can be speed up up to 2.7 times. It only takes 448.34 milliseconds (or 0.44834 seconds) to complete the calculations.

n = 100000000		
Number of Drones	**Time Taken (ms)**	**Speedup**
1 (Serial)	1200.704989	1x
2	1259.553601	0.9532781997x
4	716.7487898	1.675210347x
6	521.1246207	2.30406498x
8	448.3453012	2.678080903x

Conclusion

The results above has proven that our hypothesis is correct. For an originally time consuming task, it is desirable to use as many as drones as possible because the speed up of the computation increases as the number of available drones increase. In conclusion, HiveMind with multiple drones can provide performance improvements by speeding up the task via off-loading the work to drones.

SYSTEM ACHIEVEMENTS

- Automated deployment from a single node to other nodes, based on configuration stored in JSON files.
- Shared memory module to bypass Golang's inability to catch SEGVs
- Main functionality of the HiveMind module

- IPC

SYSTEM LIMITATIONS/SHORTCOMINGS

- Methods to handle drone failures have not been implemented (Attempts will be made before the demo)
- Methods to handle drones joining have not been implemented
- The garbage collection portion of TreadMarks has not been implemented
- The QuickSort application we wanted to use for benchmarking the system is not fully functional
- The system becomes unstable at larger computations, ie running the Pi application with 5 billion iterations causes spurious messages to form and crash drones
- Creating applications for HiveMind is somewhat cumbersome

ALLOCATION OF WORK

Dashaylan planned the overall system design, implemented the memory module and unit tests, wrote most of HiveMind data structures, handled message sending and responses at the HiveMind level such as locking and barrier protocols, integrated GoVector and ShiViz, wrote the TIPC code, Pi, Jacobi and SimpleDSM for local testing with the TIPC and drew the report diagrams. Kamil wrote the vector clocks, handled diff creation/incorporation and wrote Quicksort. Vincent wrote the deployment code, set up and maintained the Azure machines, and wrote IPC module along with Edward. Edward helped write the IPC module, wrote unit tests for various parts of the project, converted Pi to be compatible with HiveMind, performed the evaluation tests, and debugged some of the bugs.

REFERENCES

[1] Amza, C., Cox, A., Dwarkadas, S., and Kehler, P. et al. 1996. TreadMarks: shared memory computing on networks of workstations. Computer 29, 2, 18-28.

[2]Costa, M., Guedes, P., Sequeira, M., Neves, N. and Castro, M. 1996. Lightweight logging for lazy release consistent distributed shared memory. ACM SIGOPS Operating Systems Review 30, SI, 59-73.

[3]Zeng, J. 2002) Distributed Shared Memory System with Fault Detection Support.

[4]Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (WTEC'94). USENIX Association, Berkeley, CA, USA, 10-10.