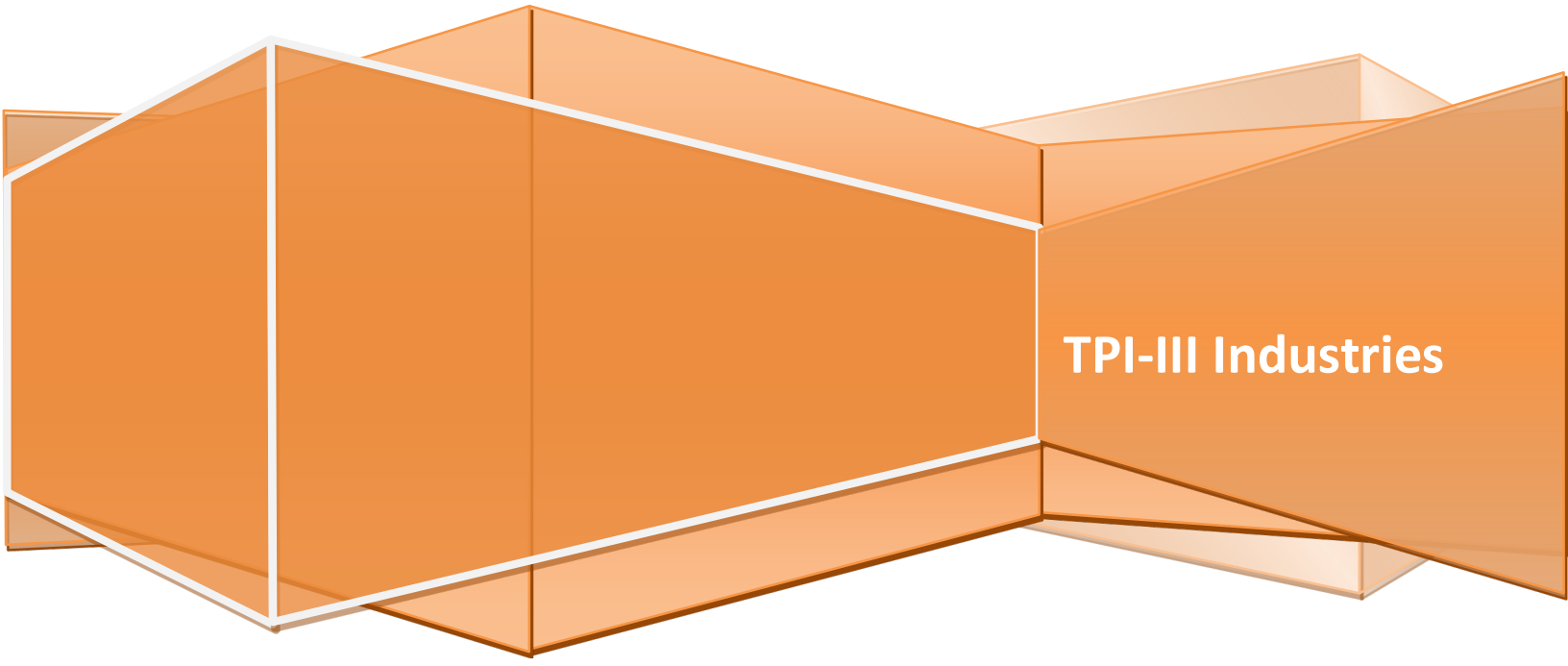
The *Hoochamacallit* System

Product Requirements Document

Thorton P. Snodgrass III



1. Overall System Idea 3
Background 3
The Team 3
Overall Considerations 3
What To Submit 4
2. Data Creator 5
Purpose 5
Details 5
3. Data Reader 6
Purpose 6
Details 6
4. Data Corruptor 9
Purpose 9
Details 9
5. DR Master List Structure 11
6. Application Logging 12
Data Creator 12
Data Reader 12
Data Corruptor 12

1. Overall System Idea

Background

The Hoochamacallit System is meant to model a network of specialized production machines (called Hoochamacallits) all communicating their current running status with a backend server. The server keeps tabs on which machines are running and which are not. This is a "Thorton Proof-of-Concept Simulator" system and as such, the server and all of the production machines will be actually running on a single Linux installation.

In order to complete this simulator, you need to create an application suite (a system) that consists of three distinct processing components:

- a "data creator" (DC) application – call the application DC. This models the production machine.
- a "data reader" (DR) application – call the application DR. This models the server.
- a "data corruptor" (DX) application – call the application DX. This acts as a stress testing module.

In this system's development, you will design and implement the data creator application, the data reader application, and the data corruptor application.

The purpose of this assignment is to force all involved in the solution to see how / why and when certain IPC mechanisms are used within UNIX / Linux System Application programming. That is, what IPC mechanism is better suited for performing a certain type of communication job over another form of IPC.

The Team

There is quite a lot of thought, design and coding in creating this system. I suggest that you find a partner to work with – but if you'd rather do it alone, you are welcome to do that as well. If you are working with a partner, I suggest that you break the work up as follows:

- one person can develop both the DC (client application) and the DX (corruptor application)
- one person can develop the DR (server application)

Before you begin on the assignment, please use the **A-03 Group** sign-up (under the *Groups* option in *Course Tools*) to organize yourselves into groups. You may have up to 3 people in your group. Remember that **all members need to sign-up** and enroll in the **same group**. Find a group quickly and start organizing, designing and coding your system! NOTE: Even if you are working alone on the system – you need to sign up for a group! Otherwise you will not see the drop-box when it comes time to submit!

Overall Considerations

Here are a couple of other things to consider while designing and implementing your solution:

- all applications must have modular design
- all applications must have file-header and function comment blocks and as always – don't forget the in-line comments within your code (these comments indicate your design thought process)
- all code developed in this project must be written in **C**
- please hand in your code in the required directory structure (for a system) with makefiles for each of the DC, DR, DX components as well as a makefile for the overall Hoochamacallit System

- Please see the Linux-Development-Project-Code-Structure document in the *Course References* area of eConestoga for information on how to structure the code for a more complex project consisting of multiple binaries
    - NOTE: In this system, the DR and DX applications should be built/copied into the same folder and the DC application must be built into its own "`bin`" folder. In the end – you should run/launch the DR and DX applications from the same "`bin`" folder and the DC application from its own "`bin`" folder

What To Submit

Since every person and/or partnership would have signed up for a team – everyone effectively has their own drop-box. So there is no need to specify the team members in the name of the submitted TAR file. Instead – because this is a larger system development, please name your submission `Hoochamacallit.tar`.

2. Data Creator

Purpose

The data creator (DC) program's job is to artificially generate a status condition representing the state of the ***Hoochamacallit*** machine on your shop's floor.

Details

- This application will send a message (via a **message queue**) to the Data Reader application (the server component of the solution).
    1. Make sure that your DC program follow best practices and checks for the existence of the message queue before sending its first message
    2. If the message queue doesn't exist, then the DC application will enter a loop in which it will sleep for 10 seconds and try again to see if the message queue has been established and created. The DC continues this sleep and check protocol until the queue exists
    3. Only after the message queue has been created can the DC application enter its main processing loop
    4. Once the existence of the queue has been established, the first message that **must be sent** is an *Everything is OKAY* type message
- Each message must contain the machine's ID value (use the PID of the process)
- Each message must contain a randomly selected status (you'll need to read up on how the *rand()* function works) value (except for the 1st message) from the list below
    1. `0` : means *`Everything is OKAY`*
    2. `1` : means *`Hydraulic Pressure Failure`*
    3. `2` : means *`Safety Button Failure`*
    4. `3` : means *`No Raw Material in the Process`*
    5. `4` : means *`Operating Temperature Out of Range`*
    6. `5` : means *Operator Error*
    7. `6` : means *`Machine is Off-line`*
- The DC will send such a message on a random time basis – between 10 and 30 seconds apart
- In making the status value of the message random as well as the frequency of the message itself, this application's output is not predictable
    1. please note that the DC **should not expect a response or reply message** – all this application needs to do send message after message
- It is expected that each DC application logs its activity to a common logging file – see the *Application Logging* section at the end of this document
- When the DC randomly selects the *`Machine is Off-line`* status and sends the message, the application may exit
    1. until the *Machine is Off-line* message is randomly generated and sent, the DC continues to send message after message
- While developing the DC application, feel free to have your debug output being printed to the screen. However, when you submit your final version of this application – there **must be no output** being printed to the screen. This application requires no input from and **no output** to the user.

NOTE: It is expected that while running and testing your system solution, you could have multiple DC applications running. Each DC represents a different Hoochamacallit machine on the shop floor. For the purposes of this system, there will be a maximum of 10 different machines that need to be supported on the shop floor.

3. Data Reader

Purpose

The data reader (DR) program's purpose is to monitor its incoming message queue for the varying statuses of the different ***Hoochamacallit*** machines on the shop floor. It will keep track of the number of different and active machines present in the system. The DR application is solely responsible for creating its incoming message queue and when it detects that all of its feeding machines have gone off-line, the DR application will free up the message queue, release its shared memory and the DR application will terminate.

Details

- This application is solely responsible for the creation and destruction of its message queue and shared memory segment
    1. When the DR application starts:
        - It will follow best practices and check for the existence of its message queue, and if not found, will create it
        - It will also create enough space in shared memory for the *master list* as indicated below
        - After allocating these resources, the DR will sleep for 15 seconds, before beginning its main processing loop (this is to give you time enough to launch some DC clients and have them begin to feed the DR with messages)
- The DR will create and maintain a ***master list*** of all clients that it has communication contact with
    1. This list will be implemented as a **structure. This structure needs to hold the following data elements**
        - The first element needs to contain the message queue ID value that the DR and DC processes are using to communicate
        - The second element will represent the number of DC processes currently communicating with the DR
        - See the *DR Master List* definition at the end of this document
    2. This *master list* must be implemented in **shared memory**
        - It will be the responsibility of the DR process to create / allocate enough space for this shared memory based upon the key_value gotten with the following function call
            - `shmKey = ftok(".", 16535);`
    3. Things to watch out for in this master-list :
        - When the DR recognizes that it hasn't heard from a DC application for more than 35 seconds, it removes it from the list as assumes that something bad has happened to that DC
            - You will need remove the DC from the `numberOfDCs` in the master list
            - You will need to remove the DC identifier (and any other information about the DC) from the list of DC clients in the master list. When you remove a DC from this list – you will collapse the information of all subsequent DCs to make sure that each remaining DC identifier is in an adjacent element of the list
                - i.e. No blank elements – see example below
            - The DR needs to log the removal of a DC in the data monitoring log as a
                `DC-YY [XXX] removed from master list - NON-RESPONSIVE`
                - (where YY is the 1-indexed ID of the DC in the master list and XXX is the PID of the DC being removed)
                - NOTE: In all logging messages given below, "YY" represents the 1-indexed ID of the DC and "XXX" represents the PID of the DC

- With each message that comes in from a DC application, the DR application will perform the **following 4 operations inside its main processing loop …**
    1. Check its master list of machines to see if this new incoming message is from a machine ID that has not been seen before
        - If it is new, then it adds the machine ID to its list
            - This event is logged as `DC-YY [XXX] added to the master list – NEW DC – Status 0 (Everything is OKAY)`
        - If this machine is known, it updates the entry for that DC in the master list to keep track of the time that this message came in – to monitor the last time the DR has *heard from* each machine
            - This event is logged as `DC-YY [XXX] updated in the master list – MSG RECEIVED – Status ZZ (aaaaaaaaa)` (where "ZZ" represents the random message sent by the DC and "aaaaaaaaa" represents the textual description of the status)
            - In the event that the message status is a value of 6 (meaning the DC has gone off-line), then the DR will instead log the event `DC-YY [XXX] has gone OFFLINE – removing from master-list`
        - **No reply message** needs to be **sent** to the sending DC process
    2. The DR will check in its master list to see if it has been more than 35 seconds since the last time it heard from ***any*** of the machines
        - If so, it can assume that that machine is dead and off-line or it exploded (you know those Hoochamacallit machines ☺) and the DR can remove that machine from its master list
            - Remember to properly collapse the master-list elements so that adjacent elements in the list still contain valid DC identifiers
            - For example :
                - If your DR is talking to 5 DC processes, your master-list would look like
                    - Master->element1 = msgQueueID value
                    - Master->element2 = 5
                    - Master->DCElements[0] = DC-01 identifier
                    - Master->DCElements[1] = DC-02 identifier
                    - Master->DCElements[2] = DC-03 identifier
                    - Master->DCElements[3] = DC-04 identifier
                    - Master->DCElements[4] = DC-05 identifier
                - Let's say DC-03 appears to have gone off-line – after properly collapsing the master list it would look like this
                    - Master->element1 = msgQueueID value
                    - Master->element2= 4
                    - Master->DCElements[0] = DC-01 identifier
                    - Master->DCElements[1] = DC-02 identifier
                    - Master->DCElements[2] = DC-03 identifier [formerly DC-04]
                    - Master->DCElements[3] = DC-04 identifier [formerly DC-05]
        - In this case, the client (DC) didn't say they were going offline, but the server (DR) detected it – in this case the assumption that the DC is dead must be logged with the "non-responsive" logging message noted above.
        - Note from the example above that the DC-## is basically given from the `DCElements` index in the array …
    3. When the DR application gets an incoming message indicating that a particular machine ID has truly gone off-line, it will remove this machine from its master list (and log an "offline" event as noted above)
        - When the number of machines registered in the master list reaches zero
            - the DR logs this event (i.e. that all Hoochamacallit machines have terminated and the DR is going to terminate) as `All DCs have gone offline or terminated – DR TERMINATING`
            - after logging this final message, the DR will remove its queue and free up its shared memory and exit

- the DR can assume that once a machine has sent an offline message, it will never receive another message from that machine

4. The final step in the DR's processing cycle is to sleep for 1.5 seconds before going back to check the message queue for another message (i.e. before going back to Step (1) above)

- As with the Data Creator – feel free to have any output being printed to the screen while you are developing and debugging this application. But when you submit the code – ensure that there is **no output** coming from this application.

4. Data Corruptor

Purpose

The data corruptor (DX) program's purpose is gain knowledge of the resources and processes involved in the application suite and then randomly decide between a set of allowable corruptions including :

- kill a DC process (to test a DC application going offline in the application suite)
- to delete the message queue being used between the set of DC applications and the DR application

If you really think about, the requirements (as listed above) for the DC and DR applications generally describe the *happy path* through the applications. The purpose of the DX application therefore is to create the alternative (or exceptions) paths through the DC and DR applications …

Details

- As soon as the DX application is launched, it will try to attach to a piece of shared memory which has been created by the DR process
  1. The key to use in order to get the shared-memory ID can be gotten from the following call
     - `shmKey = ftok(".", 16535);`
     - The DX will try to get the shared-memory ID value using this key
       - If the shared-memory piece has not yet been created by the DR, then the DX application will sleep for 10 seconds and try again.
       - This re-try will continue until either
         - 100 retries have taken place ***or***
         - The DR comes online and creates the shared-memory so that the DX gets the proper shared-memory ID
  2. <u>It is important that the DX application is running in the same directory as the DR application</u> (because of the parameters used in the `ftok()` function call)
     - <u>NOTE:</u> The DR and DX application must to run (i.e. launched) from the same directory. However, the DC application should be able to be launched from its own directory (i.e. `Hoochamacallit/DC/bin`)
  3. This piece of shared-memory will be created and maintained by the DR application. Which means that the DX application will simply be reading the information in this area of shared memory to gain knowledge of necessary information
     - This area of shared-memory is maintained as a ***master list*** of all clients that the DR is in communication contact with
     - This list will be implemented as indicated in the *DR Master List* definition at the end of this document
- The DX application's main processing loop will be comprised of the **following 4 steps**:
  1. Sleep for a random amount of time (between 10 and 30 seconds)
  2. When it awakes, the DX should check for the existence of the message queue (between the DC's and the DR)
     - If the message queue no longer exists, the DX assumes that all of the DC's have shut down and the DR (having detected this) has exited
     - The DX will then log this event as follows, detach itself from shared memory and exit itself

```
[2020-03-06 21:05:07] : DX detected that msgQ is gone – assuming DR/DCs done
```

  3. Select an action (from the ***Wheel of Destruction*** below) randomly
     - For the purposes of logging, the ***Wheel of Destruction*** will be abbreviated to ***WOD***

```
00 : do nothing
01 : kill DC-01 (if it exists)
02 : kill DC-03 (if it exists)
```

```
03 : kill DC-02 (if it exists)
04 : kill DC-01 (if it exists)
05 : kill DC-03 (if it exists)
06 : kill DC-02 (if it exists)
07 : kill DC-04 (if it exists)
08 : do nothing
09 : kill DC-05 (if it exists)
10 : delete the message queue being used between DCs and DR[1]
11 : kill DC-01 (if it exists)
12 : kill DC-06 (if it exists)
13 : kill DC-02 (if it exists)
14 : kill DC-07 (if it exists)
15 : kill DC-03 (if it exists)
16 : kill DC-08 (if it exists)
17 : delete the message queue being used between DCs and DR[1]
18 : kill DC-09 (if it exists)
19 : do nothing
20 : kill DC-10 (if it exists)
```

4. Execute the action using whatever system call, IPC call, etc. you need to in order to get the job done
    - When "killing" one of the DC processes – make sure to send it a **`SIGHUP`** signal

- Please refer to the Application Logging section below for additional information on the DX's logging responsibilities
- Also keep in mind that since this application is really an *intruder* application – that when trying to kill processes, delete files or remove message queues
    1. Proper error checking and handling ***had better be done***. So that the DX application doesn't crash!
    2. For example – what if the DX is about to send a "kill" message to a DC, but the DC exits in the meantime … how with the DX handle the failure of the "kill" command?
- In the final submitted version of the DX application, there should be **no output** to the screen

[1] NOTE: when the message queue disappears, it is expected that all applications (DC, DR and DX) will exit

5. DR Master List Structure

```
#define    MAX_DC_ROLES     10

typedef struct
{
      pid_t       dcProcessID;
      ???         lastTimeHeardFrom;
} DCInfo;

typedef struct
{
      int         msgQueueID;
      int         numberOfDCs;
      DCInfo      dc[MAX_DC_ROLES];
} MasterList;
```

As you can see from the above structure, you are left to determine an appropriate datatype to use for the *lastTimeHeardFrom* data element.

6. Application Logging

Data Creator

- Here is a little more information about the formatting of the required DR logging messages:
    1. Every log event captured is written to a standard ASCII log file – to be found in the following file: `/tmp/dataCreator.log` (NOTE: the log file is being written to the *tmp slice* (as I call it in class))
    2. Each log entry must start with a timestamp (current date and time). (You may find *`asctime()`* a useful function to work with (prototyped in `time.h`), along with *`localtime()`* as a function to query the system for the current local time)
    3. Each log entry will be created and written after the status message is sent to the DR
    4. Each message consists of the DC's PID, the random status generated as well as a textual description. For example – the log entry for a DC (with PID=5687) generates a status value of 2. The message would look like this:

```
[2020-03-06 21:05:07] : DC [5687] – MSG SENT – Status 2 (Safety Button Failure)
```

Data Reader

- Here is a little more information about the formatting of the required DR logging messages:
    1. Every log event captured is written to a standard ASCII log file – to be found in the following file: `/tmp/dataMonitor.log` (NOTE: the log file is being written to the *tmp slice* (as I call it in class))
    2. Each log entry must start with a timestamp (current date and time)
    3. Each log entry consists of the required message (as indicated in Section 3 above)
    4. Feel free to log any additional information that you feel would be beneficial within the log file (i.e. think about what events or actions that the DR takes would be beneficial to log)
- An example log entry appears as follows

```
[2020-03-06 21:05:07] : DC-02 [5687] updated in the master list – MSG RECEIVED – Status 2 (Safety Button Failure)
```

Data Corruptor

- Over and above this main processing loop, the DX application is responsible to logging each action that it takes in a standard ASCII log file – to be found in `/tmp/dataCorruptor.log` (NOTE: the log file is being written to the *tmp slice* (as I call it in class))
    1. Each log entry must start with a timestamp (current date and time)
    2. After the message timestamp, the DX will log some textual description of what has happened
        - For example : If action 01 was randomly selected then log a message saying something like :

```
[2020-03-06 21:05:07] : WOD Action 11 – DC-01 [7565] TERMINATED
```

    3. If the DX randomly chose option 10 (the deletion of the message queue), then the DX logs the following event (as follows) and then detaches itself from shared memory and exits

```
[2020-03-06 21:05:07] : DX deleted the msgQ – the DR/DCs can't talk anymore - exiting
```