**An Implementation of the TEMPO Berkeley Algorithm and Election Algorithm For Distributed Clock Synchronization**

Jordan Sturtz

Dr. Siddula

COMP 755 - Advanced Operating Systems

23 Nov. 2021

# **Introduction**

The Berkeley algorithm was designed by Gusella and Zatti and implemented at the University of Berkeley on UNIX machines (1983; 1985a; 1985b; 1987). My project is an implementation of the Berkeley clock synchronization algorithm together with their proposed election algorithm. The Berkeley algorithm is a clock-synchronization algorithm designed to keep nodes in a distributed system in sync with each other across an intranet. The algorithm as they describe it is a client-server architecture, where one machine designated the “leader” periodically queries the other nodes in a distributed system to get their current times, and then the leader computes an average time and requests that every node in the system adjust their clocks to match this average time (Figures 1 and 2). Though this design pattern is fundamentally a client-server architecture, Gusella and Zatti also designed and implemented an election algorithm (1985b) to make their clock synchronization approach tolerant to node failures (Figure 3). The nodes in the system elect amongst themselves a single node to be the leader and the rest become followers. There is no priority ranking of these nodes or any attempt to synchronize time in the nodes to some node outside the intranet, unlike in the Network Time Protocol used in the Internet (Mills, 1991).

The goal of my project is to implement both the Berkeley TEMPO algorithm for synchronizing clocks and the corresponding election algorithm to handle electing new leaders. Though the election algorithm adds extra complexity to the design, implementing it correctly helps to create *failure transparency* for the distributed system. If a single node fails or if the intranet is partitioned in such a way as to cut off a leader from the rest of the nodes, it will automatically adjust to elect a new leader. No special intervention should be needed by the users of this distributed system.

To create the environment for my project, I used Docker and Docker-Compose to handle spinning up separate containers running instances of the Ubuntu distribution of linux. Each container has a time-daemon.py script with the same program text. Each running instance of time-daemon.py communicates over the docker network to pass messages to implement the Berkeley algorithm and the corresponding election algorithm. To simulate reporting and correcting local clock times, I used an online library called libfaketime created by wolfcw (2021) and published for general use in GitHub.

| ***Figure 1***    *Note*. Taken from Gusella and Zatti (1987) | ***Figure 2*** |
| --- | --- |

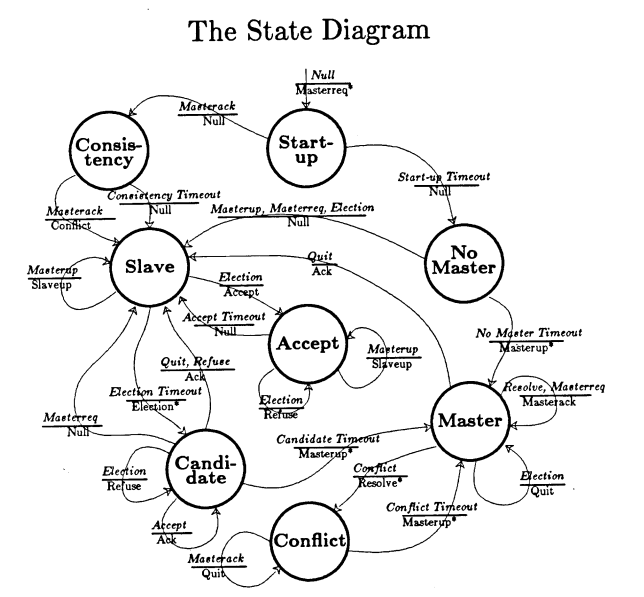
# **Related Research on Clock Synchronization**

# Gusella and Zatti published four papers for their implementation of the Berkeley algorithm (1983; 1985a; 1985b; 1987). Their earliest paper in 1983 describes the TEMPO client-server architecture. They published two papers in 1985, one that describes their Time Synchronization Protocol (TSP) (1985a) for message passing between nodes and another that describes their election algorithm for electing a new leader (1985b). Finally, they published a paper on the efficiency of the Berkeley algorithm in 1987.

There are many different types of clock synchronization protocols (Tanenbaum, 2007). One of the earliest proposals was a proposal by Cristian (1989) that, like the Berkeley algorithm, proposes a client-server architecture. Unlike the Berkeley algorithm, however, in Cristian’s proposal the clientcontacts the server to request an update to its own clock rather than in the Berkeley algorithm where the server periodically queries the clients.

Another important clock synchronization approach is the Network Time Protocol (NTP) designed by Mills (1991). NTP is used to synchronize clocks of machines connected to the Internet, and it is known to maintain worldwide accuracy of clocks connected to the internet within 50ms (Tanenbaum, 2007). An important difference between NTP and other clock synchronization algorithms is that NTP is designed to synchronize clocks some objectively correct measure of time. As such, the NTP relies on a categorization of nodes in the system into levels or strata of accuracy, where those of the lower strata contact nodes of higher strata to request more accurate times. There is no such asymmetry with the Berkeley algorithm, as no node in the system is given any priority in terms of deciding the correct time.

***Figure 3***



*Note*. Taken From Gusella and Zatti (1985b)

The Reference Broadcast Synchronization Protocol (RBS) designed by Elson et al. (2012) is similar to the Berkeley algorithm in that its goal is to keep only the nodes in a system in sync with each other. Unlike the Berkeley algorithm, however, RBS keeps only *receivers* in sync and does not attempt to synchronize the clock of the *sender* of messages over a network (Tanenbaum, 2012).

Another related approach to solving the problem of synchronization in a distributed system is that rather than trying to synchronize clocks, we instead simply maintain a correct ordering of events across cooperating processes in a distributed system (Tanenbaum, 2012).. Indeed, if the goal of an application is merely to keep events ordered correctly, then clock synchronization may be too strong of a requirement. It is for this reason that Lamport (1978) developed what has been termed “logical clocks” as a method for synchronizing processes by guaranteeing that the order of events is preserved across cooperating processes.

There does not appear to be many papers on related projects implementing the Berkeley algorithm. Sehgal (2007) wrote a paper that appears to be unpublished that compares the performance of Cristian’s algorithm against the Berkeley algorithm. Sehgal used something called Simulation Engine, which they did not describe in detail, to generate transmission delays and synchronization errors. Their results are hard to interpret and their graphs appear to be incomplete.

# **Interface**

To create the environment for testing a distributed system, I used Docker (<https://docs.docker.com/engine/reference/builder/>) and Docker Compose (<https://docs.docker.com/compose/>). I wrote a *Dockerfile* that uses an existing Ubuntu image from DockerHub and extends it by adding necessary dependencies such as python, cmake, libfaketime and also by copying files from the host machine to each docker container. My Dockerfile also automatically executes *make* on the installed *libfaketime* code to allow me to simulate setting clock times and drifts.

The third-party package *libfaketime* (<https://github.com/wolfcw/libfaketime>) allowed me to get around a limitation of the Docker Containers, which is that the system call to *hwclock* did not appear to work. So instead, my processes as they run will set an operating system environment variable that will allow *libfaketime* to intercept system calls and simulate a response back to the caller. In this way, I was able to set a clock offset and drift. In figure 4, for example, I set the environment variable “FAKETIME” equal to “+1y x2”, i.e. report back a time that is one year in the future and set the clock drift equal to twice the normal speed.

**Figure 4**

# 

**Software Implementation**

# My project is written in python. Once the docker-compose process has spun up containers, a user of this application currently has to use an interactive bash shell inside each docker container to start the process. The docker image will install my script into /lib/time-daemon/time-daemon.py. No command line arguments are needed.

The design of the program is simple and follows Gusella and Zatti’s finite state model referenced in figure 2. I have written two classes, one the TimeDaemon class that represents the time daemon itself and another a Messenger class that handles message passing. At the entry point for my program, I create an instance of my TimeDaemon class and call its run method:

# ============================================================================

def main():

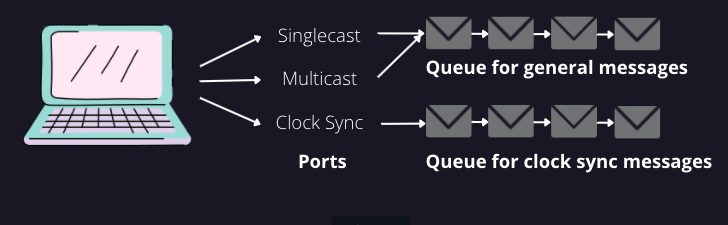
TimeDaemon().run()

if \_\_name\_\_ == "\_\_main\_\_":

main()

The constructor of the TimeDaemon class first sets the clock drift on the machine by randomly picking a value between 1 - MAX\_CLOCK\_DRIFT and 1 + MAX\_CLOCK\_DRIFT. It then creates an instance of the messenger class to be used in its own methods. Finally, it creates three listener threads that will listen for messages on three different ports, forwarding those messages into two message queues, as illustrated in figure 5.

**Figure 5**



The code for the constructor is below:

class TimeDaemon():

def \_\_init\_\_(self) -> None:

# Once we've gotten a random drift, update the faketime to have that drift

os.environ["FAKETIME"] = f"+0.00s x{MY\_CLOCK\_DRIFT}"

self.my\_ip\_address = socket.gethostbyname(socket.gethostname())

self.\_\_state = State.START

self.\_\_messenger = Messenger(self.my\_ip\_address)

self.singlecast\_port = 1000

self.singlecast\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

self.singlecast\_sock.bind((self.my\_ip\_address, self.singlecast\_port))

self.clock\_sync\_port = 1001

self.clock\_sync\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

self.clock\_sync\_sock.bind((self.my\_ip\_address, self.clock\_sync\_port))

# Create a multicast socket for listening for / sending to multiple receivers

self.multicast\_port = 10000

self.multicast\_group = ('224.3.29.71', 10000)

self.multicast\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

self.multicast\_sock.bind(('', self.multicast\_port))

# Tell the operating system to add the multicast socket to the multicast group

group = socket.inet\_aton(self.multicast\_group[0])

mreq = struct.pack('4sL', group, socket.INADDR\_ANY)

self.multicast\_sock.setsockopt(

socket.IPPROTO\_IP, socket.IP\_ADD\_MEMBERSHIP, mreq)

# We will direct traffic from the singlecast sock and multicast sock to the

# same queue

self.\_\_message\_queue: queue.Queue = queue.Queue()

self.singlecast\_listener = self.\_\_messenger.create\_listener\_thread(

self.singlecast\_sock, self.\_\_message\_queue)

self.singlecast\_listener.start()

self.multicast\_listener = self.\_\_messenger.create\_listener\_thread(

self.multicast\_sock, self.\_\_message\_queue)

self.multicast\_listener.start()

# We will also use a separate queue for clock sync to simply processing

self.\_\_clock\_sync\_queue: queue.Queue = queue.Queue()

self.clock\_sync\_listener = self.\_\_messenger.create\_listener\_thread(

self.clock\_sync\_sock, self.\_\_clock\_sync\_queue)

self.clock\_sync\_listener.start()

The run() method starts the process of looping infinitely and changing states:

# =============================================================================

def run(self) -> None:

while True:

print(self.\_\_state)

if self.\_\_state == State.START:

self.start()

elif self.\_\_state == State.CONSISTENCY:

self.consistency()

elif self.\_\_state == State.NOLEADER:

self.no\_leader()

elif self.\_\_state == State.FOLLOWER:

self.follower()

elif self.\_\_state == State.ACCEPT:

self.accept()

elif self.\_\_state == State.CANDIDATE:

self.candidate()

elif self.\_\_state == State.LEADER:

self.leader()

elif self.\_\_state == State.CONFLICT:

self.conflict()

Then, when a time-daemon.py is in a particular state, it will handle listening for messages and changing states as needed, according to the logic laid out in Gusella and Zatti’s state diagram (Figure 2). For instance, the start state looks like this:

# =============================================================================

# STATE FUNCTIONS START HERE

# =============================================================================

def start(self) -> None:

# Randomly assign a timeout so followers are less likely to elect themselves

timeout = random.randrange(RESYNC\_RATE, RESYNC\_RATE \* 2)

# Then, send out a request for a leader to respond

self.send\_multicast\_signal(Signal.LEADERREQ)

# Wait here until up to timeout for the response

message = self.wait\_for\_signal\_from\_queue(

self.\_\_message\_queue,

Signal.LEADERACK, timeout)

# If the signal was detected, there's a leader

if message:

self.\_\_first\_leader\_ip: str = message["ip\_address"]

self.\_\_state = State.CONSISTENCY

else:

self.\_\_state = State.NOLEADER

Once a leader has been chosen, it sends a request to all the known followers to ask them to report their current clocks. After waiting for a period of time, the leader then computes the Delta for each follower and an average among all the followers. In Gusella and Zatti’s implementation, however, they also designate a cutoff for how bad a clock must be to exclude it from the mean computation. I did not implement this because in my case, my implementation has fairly large differences at the outset due to *libfaketime.* Once the average has been computed, the Leader node fixes its own clock to match this average time and then the Leader computes the difference between the average and the reported times of each follower to send a request to each to adjust their clocks accordingly. See figures 1 and 2 for an illustration.

The fix\_my\_clock method below shows how I updated the os environment variable used in libfaketime to report times:

# =============================================================================

def fix\_my\_clock(self, adj\_seconds: float) -> None:

adj\_seconds = round(adj\_seconds, 3)

print(f"Adjusting my clock by {adj\_seconds} seconds")

# Implement fixing the local clock

current\_offset\_as\_str, current\_drift = os.environ["FAKETIME"].split(" ")

# Ignore sign and second symbol, convert to float

current\_offset = float(current\_offset\_as\_str[:-1])

# Add the adjustment

current\_offset += adj\_seconds

# Reassign back to env variable

os.environ["FAKETIME"] = "{:+.{precision}f}s {}".format(

current\_offset, current\_drift, precision=SEC\_DECIMAL\_ROUND)

The below snippet shows the method that the Leader uses to perform the clock synchronization mean computation and estimates for clock adjustments:

# -----------------------------------------------------------------------------

while True:

quit\_event.wait(timeout)

if quit\_event.is\_set():

break

with follower\_lock:

if followers:

follower\_clock\_diffs = get\_follower\_diffs(followers)

# Log the diffs before fixing

log\_follower\_diffs(follower\_clock\_diffs)

# Add a 0 for the leader so average works properly

avg\_diff: float = mean(

list(follower\_clock\_diffs.values()) + [0.0])

# We adjust our own clock to the avg diff

self.fix\_my\_clock(round(avg\_diff, SEC\_DECIMAL\_ROUND))

# We adjust followers' clocks to

# avg\_diff - their estimated drift

for follower, diff in follower\_clock\_diffs.items():

message\_dict = {

"signal": Signal.SYNCREQ,

"adj\_seconds": avg\_diff - diff,

}

self.send\_message\_to\_ip\_address(

message\_dict,

follower,

self.clock\_sync\_port,

self.clock\_sync\_sock)

# Log the diffs after fixing

log\_follower\_diffs(get\_follower\_diffs(followers))

The method log\_follower\_diffs is written to append information to a CSV file to report those differences to measure my performance:

# ------------------------------------------------------------------------

def log\_follower\_diffs(follower\_diffs):

leader\_time = str(time.time())

follower\_times = [str(x) for x in follower\_diffs.values()]

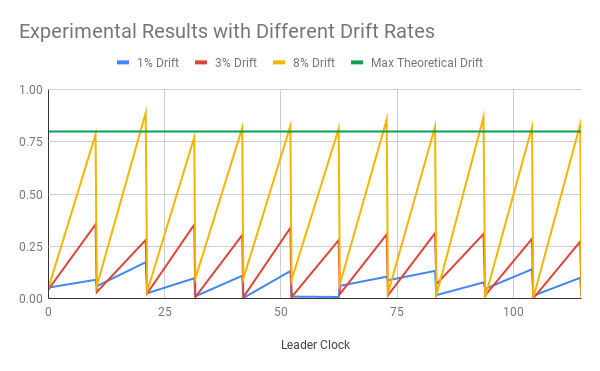
row = ",".join([leader\_time] + follower\_times) + "\n"

with open("logfile.csv", "a+") as f:

f.write(row)

Once I had those results, I used them in my Experimental Results Excel document to produce a graph to show whether it worked. See Figure 6 for those results. The green line shows the maximum drift in seconds we would expect from the leader, since the worst drift percentage from Leader in this experimental run was 8%. We can see that periodically, the time difference is corrected back down to Leader.

**Figure 6**



**Conclusion**

I did not work in a group. I had started working in a group with James Adams and Angeline Quaye, but I felt that they were not contributing fairly to the project. Before I left the group, the only portion of the project that was publicly available to all members was my docker setup. I did the work to research how to use docker, I wrote the Dockerfile and docker-compose.yml, and I wrote the README to explain how to use docker to the other group members. None of my code in time-daemon.py was at any point publicly available to them.

My main difficulty was the use of *libfaketime.* Since the best I could do was set the currently running process’s internal environment variables, I was not actually setting clock times or drifts on each virtual machine. This caused a problem where the initial clock offsets when each node starts up can diverge by a large margin from the rest of the nodes. However, as each process on each node ran, eventually those clock times were fixed after a few cycles, which enabled me to collect useful experimental results.

A second challenge is that I had to use python’s *threading* library to create asynchronous code to handle the logic of the program. This increased the complexity of the program and led to bugs that were hard to understand. My code is still buggy and will occasionally hang in different states.

If I had more time, I would have liked to find a way to compare the performance of this implementation to others. Of course, I had not attempted to optimize this implementation and it’s written in a high-level language, so such comparisons are likely unrealistic to real-world applications anyway. A poor performing program might only show poor performance due to poorly optimized logic or quirks of the high-level language itself. Moreover, comparisons to other clock synchronization algorithms might be like comparing apples to oranges. The Berkeley algorithm is designed only to synchronize clocks to each other, so can only rightfully be compared to for example the Reference Broadcast Protocol which is similar.

**References**

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed computing*, *3*(3), 146-158.

Docker Inc (n.d.). *Overview of Docker Compose*. Docker Docs. <https://docs.docker.com/compose/>

---. (n.d). *Dockerfile Reference.* Docker Docs. <https://docs.docker.com/engine/reference/builder/>

Elson, J., Girod, L., & Estrin, D. (2002). Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, *36*(SI), 147-163.

Gusella, R., & Zatti, S. (1983). *Tempo: A network time controller for a distributed berkeley unix system*. Computer Science Division, University of California.

---. (1985a). *The berkeley unix 4.3 bsd time synchronization protocol*. CALIFORNIA UNIV BERKELEY COMPUTER SCIENCE DIV.

--- (1985b). *An election algorithm for a distributed clock synchronization program*. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

--- (1987). *The accuracy of the clock synchronization achieved by tempo in Berkeley Unix 4.3 BSD*. CALIFORNIA UNIV BERKELEY COMPUTER SYSTEMS RESEARCH GROUP.

Mills, D. L. (1991). Internet time synchronization: the network time protocol. IEEE Transactions on communications, 39(10), 1482-1493.

Sehgal, R., 2007. Simulation Engine for Analysis and Comparison between Cristian’s and Berkeley clock synchronization algorithms. Project Report. Department of Computer Science, Kent State University, Kent.

Tanenbaum, A. S., Steen, M. van, & Steen, M. van. (2007). *Distributed systems : principles and paradigms* (2nd ed.). Pearson Prentice Hall.

wolfcw (2021). LibFakeTime, GitHub repository, <https://github.com/wolfcw/libfaketime>