Large-Scale Distributed Systems
Project 3: Firefly-inspired synchronization

Laurent Hayez

December 16, 2015

Table of contents

1 Introduction

In nature, fireflies produce light in order to attract mates or prey. One interesting feature of these beetles is that when they emit light in group, at some point, they do it in a synchronized manner, just by looking at when their neighbours emit light. This feature is interesting in large-scale distributed systems, as synchronization might be required, but one node does not know every other nodes.

The objective is thus to inspire ourselves from fireflies to try to synchronize nodes in a decentralized manner. At first we will detail the protocol skeleton and explain how the core of the protocols will work. Then we will look at a two models called "phase-advance" and "phase-delay" and briefly analyze them. The main and final part will be the "adaptive Ermentrout model" which is more representative of the reality. We will explain the implementation specifities and analyze this model in different situations.

2 Protocol skeleton

According to the paper "Firefly-inspired Heartbeat Synchronization in Overlay Networks", the skeleton for the different algorithms is composed of two main functions,

namely ACTIVETHREAD and PASSIVETHREAD. We provide the pseudo code for the implementation in Algorithm 2.1.

In the different protocols, a node is an oscillator characterized by its phase φ and the cycle length Δ. We define φ as a sawtooth function with domain $[0, 1]$ such that $\frac{d\varphi}{dt} = \frac{1}{\Delta}$. This is represented in Figure 1.

When φ reaches 1, the node will send a flash to a set of neighbour nodes, and φ is reset to 0. The cycle length, depending on the model chosen, can be the same or different for all nodes. The function UPDATEPHI will differ in our implementations, but we will come back on this when needed.

The core of the synchronization protocol is the function PROCESSFLASH, i.e. what a node does when it receives a flash. This function is responsible of how φ is updated. Depending on the implementation, φ will be updated, or Δ will be updated and will affect φ.

The underlying overlay network protocol used for a node to know its neighbours is the Peer Sampling Service (PSS).

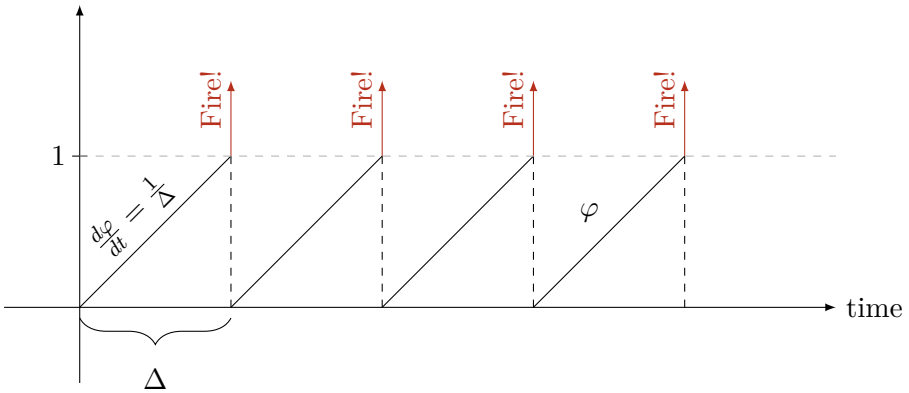

Figure 1: Representation of φ and its relation with Δ

We briefly discuss how the protocol skeleton works before we move to the implementation of the phase-advance and phase-delay protocols.

For every implementations, we start by initializing the PSS at each node, and wait two minutes to have a consistent view. Then we start a thread ACTIVETHREAD and a periodic thread UPDATEPHI. As we defined φ to be such that $\frac{d\varphi}{dt} = \frac{1}{\Delta}$, we have $\varphi = \int \varphi dt$, which explains how we update φ. If $\varphi \geq 1$, we fire the event "Flash!" that triggers ACTIVETHREAD to send that it emitted a flash to the nodes in its view. In UPDATEPHI, we then set φ to 0, as in Figure 1 and call ACTIVETHREAD to wait for a new flash.

3 Phase-advance and phase-delay protocols

3.1 Implementation

The fundamental difference with the protocol skeleton here is the implementation of the PROCESSFLASH function. As the two protocols are almost the same, we added the boolean PHASE_ADVANCE to signify the usage of the phase-advance or phase-delay protocol. The pseudo-code of the implementation is given in Algorithm 3.1.

Algorithm 2.1 Skeleton for the Firefly algorithms

Variables:

φ ▷ phase

Δ ▷ cycle length

$$\text{update\_phi\_period} = \begin{cases} \frac{\Delta}{10} & \text{if } \Delta < 1 \\ \frac{1}{10\Delta} & \text{if } \Delta \geq 1 \end{cases}$$

function sendFlash()
    $P \leftarrow$ view from PSS
    send flash to all peers in P
end function

function processFlash()
    depends on the implementation
end function

function updatePhi()
    if $\varphi < 1$ **then**
        $\varphi \leftarrow \varphi + \frac{1}{\Delta} \cdot$update_phi_period
    else
        fire event "Flash!"
        $\varphi \leftarrow 0$
        start new thread activeThread
    end if
end function

function activeThread()
    wait for the event "Flash!"
    sendFlash()
end function

function passiveThread()
    receive flash
    processFlash()
end function

Algorithm 3.1 processFlash for the phase-advance and phase-delay protocols

Variables:

phase_advance $\leftarrow$ true or false

function processFlash()
    if phase_advance **then**
        $\varphi \leftarrow 1$
    else
        $\varphi \leftarrow 0$
    end if
end function

3.2 Analysis of the protocols

We tried our implementation with 100 nodes per experiment and $\Delta = 1, 2$ and 5. For the PSS, we used a view of 10 peers with a period of 20 seconds for each update. We used a random selection for the peer selection, and we used a swapping parameter of 3 and a healing parameter of 2 for the view exchange policy.

We start by analyzing the convergence of the phase-advance protocol. On Figure 2, we can see that from time to time, the protocol seems to synchronize, it does not converge to a stable state where all the nodes emit flashes at the same time. In fact, what happens is what we see on Figures 3 and 4. Whenever a node flashes, it tells ten other nodes, that set φ to 1, emit a flash, tell ten other nodes, which emit a flash and so on. We thus have an exponential growth of the flash rate which makes the memory used explode very fast. In the logs, we had this very error:

```
17:37:43.276437 (66) Too much memory used:  wants 2097182 (max:
2097152), end of process.
```

This model is therefore highly impratical because it overloads the network very quickly. By analogy with fireflies, at first a few fireflies would emit light, and then by observing the neighbours they would emit light every time they see a neighbour emitting light, quickly resulting in emitting light constantly, which does not happen in reality.

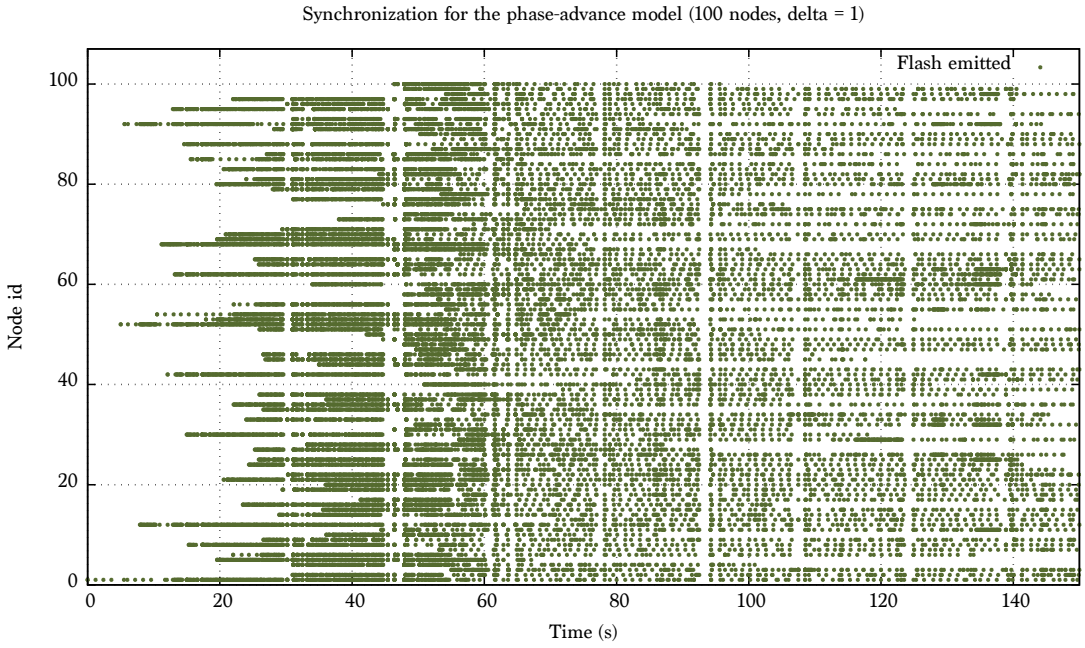

Figure 2: Phase-advance model synchronization with $\Delta = 1$

Now let's take a look at the phase-delay model. On Figures 5 and 6, we representend the emissions for $\Delta = 1$ and $\Delta = 2$ respectively. On both figures we cannot distinguish a convergence in the flash emissions. On Figure 7 however, we can see that from 150 seconds after the first flash, the nodes that emit flashes are synchronized.

Although we seem to obtain some synchronization for $\Delta = 5$, the phase-delay protocol is not guaranteed to converge to a stable state where all the nodes are synchronized. Both phase-advance and phase-delay protocols rely on the assumption that when a message is sent, it is instantly received. In practice, this assumption is highly unrealistic

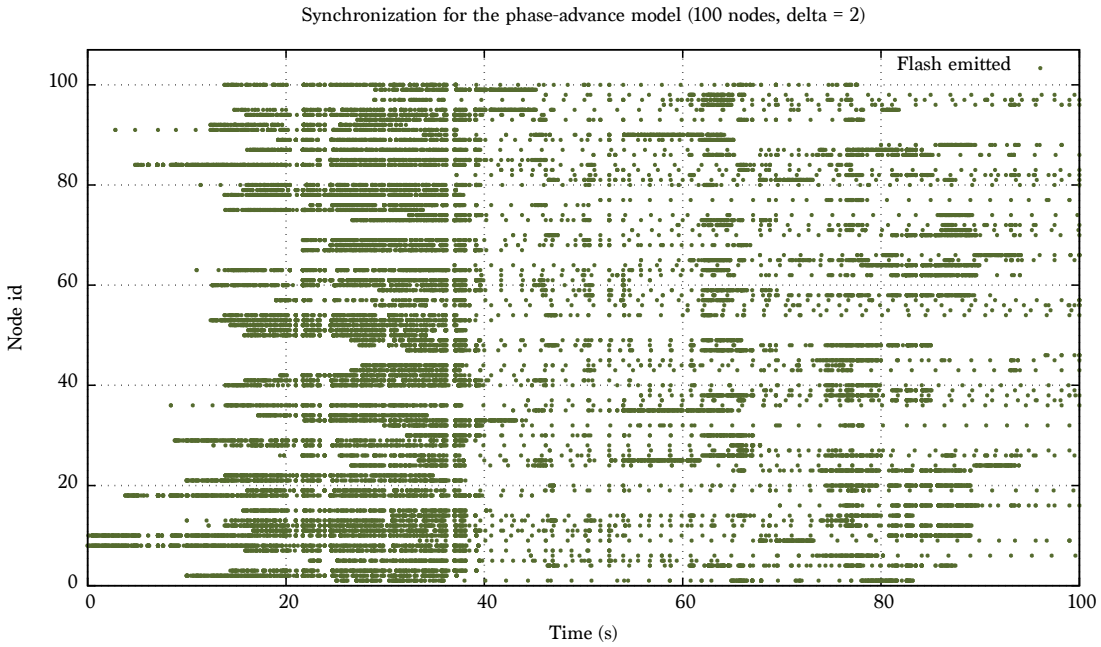

Figure 3: Phase-advance model synchronization with $\Delta = 2$

due to packet loss, congestion or simply because even the fastest networks, a message is not instantly delivered.

In our experiments, we considered Δ to be fixed and the same at all nodes. This assumption does not represent reality either; why would all fireflies emit light at the same rate? If we had different Δs at all nodes, or if the skew of the clocks is important, none of the models is guaranteed to converge, and it is rather logical as we don't update Δ. If a node starts with $\Delta = 4.5$ and another with $\Delta = 5.5$, they will never synchronize. Therefore, both phase-advance and phase-delay are mathematically insufficient to model the reality.

4 Adaptive Ermentrout model

According to our preceding conclusion, we now consider a model that allows the node to initally have different cycle lengths. This model is called the "adaptive Ermentrout model". In this model, we denote by i the i-th node in the system. Each node has an initial cycle length δ_i and a natural cycle length Δ which are bounded below and above, ie $\delta_i, \Delta \in (\Delta_l, \Delta_u)$. If there is no interaction with other nodes, a node will flash every Δ seconds.

Furthermore, we do not express the model in terms of the cycle length, but rather in terms of the frequency $\omega_i = \frac{1}{\delta_i}$. We have $\Omega = \frac{1}{\Delta}$, $\Omega_l = \frac{1}{\Delta_u}$ and $\Omega_u = \frac{1}{\Delta_l}$.

4.1 Implementation

The main difference with the phase-advance and phase-delay models is that the function processFlash does not update φ, but the frequency ω_i. If a flash arrives when $\varphi < \frac{1}{2}$ (too late), the frequency will be decreased, and if a flash arrives when $\varphi > \frac{1}{2}$ (too early), the frequency will be augmented, meaning that a node will ajust its frequency according to the flashes it receives. In addition to that, we add a new parameter ε

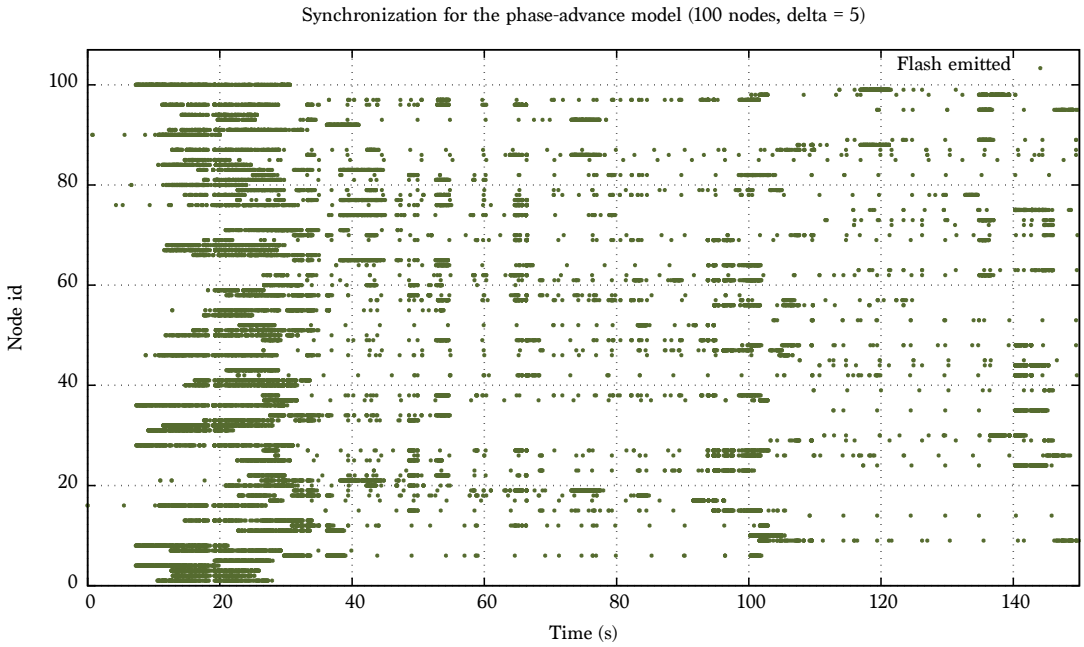

Figure 4: Phase-advance model synchronization with $\Delta = 5$

which we set to 0.01 and which represent the offset of the frequency when we update it. The pseudo-code of the algorithm is given in Algorithm 4.1 (note that the pseudo-code given is what to add/modify from the skeleton protocol).

The functions g^+ and g^- formalize the idea that if a flash arrives "too late" (respectively "too early"), we must deacrease the frequency (respectively increase it). Indeed we have that

$$g^+(\varphi) = \begin{cases} 0 < a \leq \frac{1}{2\pi} & \text{if } \varphi < \frac{1}{2} \\ 0 & \text{if } \varphi \geq \frac{1}{2} \end{cases}, \quad g^-(\varphi) = \begin{cases} 0 < a \leq \frac{1}{2\pi} & \text{if } \varphi > \frac{1}{2} \\ 0 & \text{if } \varphi \leq \frac{1}{2} \end{cases}.$$

4.2 Analysis of the protocol

The setup for this experiment was a network of 64 nodes. We used $\Delta_l = 4.5$ sec, $\Delta_u = 5.5$ sec, $\Delta = 5$ sec and δ_i a random number between Δ_l and Δ_u. The settings for the peer sampling service are the same as for the previous experiment. To plot the figure in this section we used the Python scripts `Parser-synchronization.py` and `Parser-ultimate.py`, and the bash scripts `Parse-and-plot-sync.sh` and `Parse-and-plot-ultimate.sh`. The bash scripts call the python scripts and the gnuplot script to plot the graphs.

On Figure 8, we represented how the node synchronize under the adaptive Ermentrout model. We clearly see that after 50 seconds, when most of the nodes joined the network, the nodes reaches a stable state where the emit light at the same time.

On Figure 9, we represented the emission window length according to the time. As expected, the emission window length reduces as time increases, which indicates that the algorithm converges to a stable state. There are some fluctuations however, but the general trend is to decrease and converge to somewhere between 0.5 and 1 second. On Figure 10, we plotted the cycle length for each node according to the time. The fluctuations can be explained with the cycle length, because we see that when there are

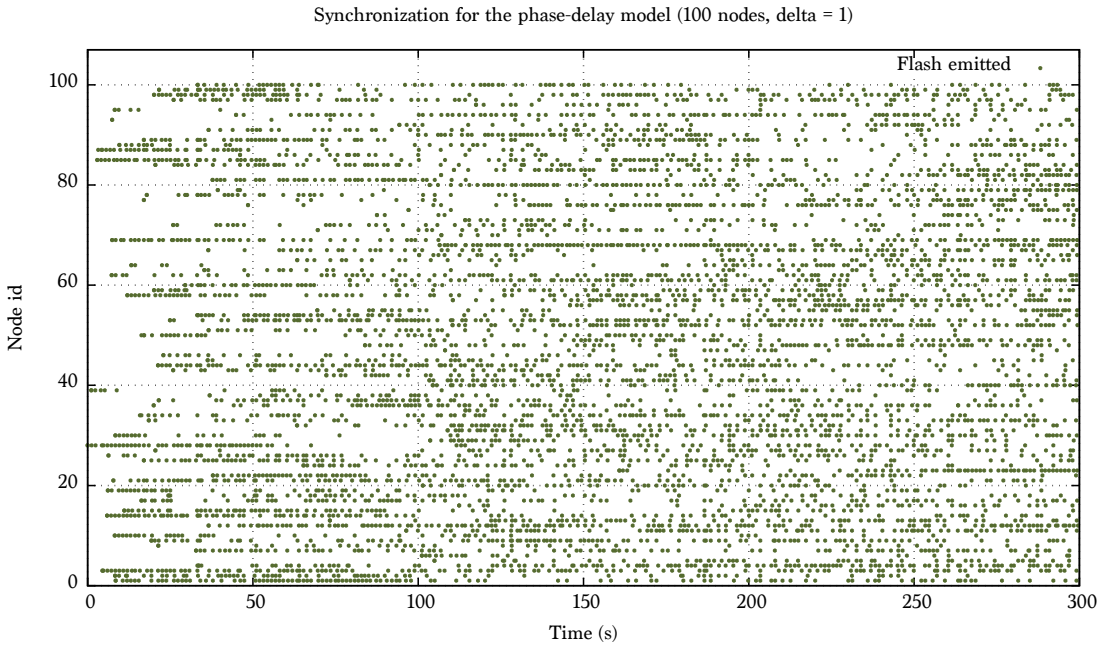

Figure 5: Phase-delay model synchronization with $\Delta = 1$

fluctuations for the emission window length, we have some weird values for the cycle lengths, explaining the occasional desynchronization. This could be because the nodes on the cluster were overloaded at this time.

4.3 Analysis of the protocol under churn

We now analyze the protocol's behaviour under churn. On Figure 11, we show the flashes the node emitted. We see that it converges until there is churn, and that under churn, the remaining nodes are not desynchronized too much. The desynchronization happens when the new node join at around 300 seconds.

The observations above are confirmed with Figure 12 that represents the emission window length under churn. We clearly see that the protocol synchronizes until the churn starts. At this point the emission window length is a little bit higher. When new nodes join the system there is a peek for the emission window length, and we see that the problem is not when node leave the system, but rather when new nodes join the system.

We clearly see this in Figure 13, which represents the cycle length under churn. Until there is churn, the cycle lengths are all around 5 seconds, but when there is churn everything starts to break, and nothing recovers.

5 Conclusion

This project inspired from fireflies to create a system that can synchronize itself in a decentralized manner, only by getting partial information of the global system from the neighbours. One important piece is the peer sampling service, that allows every node to have a partial view, but the main focus was on the mathematical models that permitted to synchronize the nodes like fireflies synchronize to emit light.

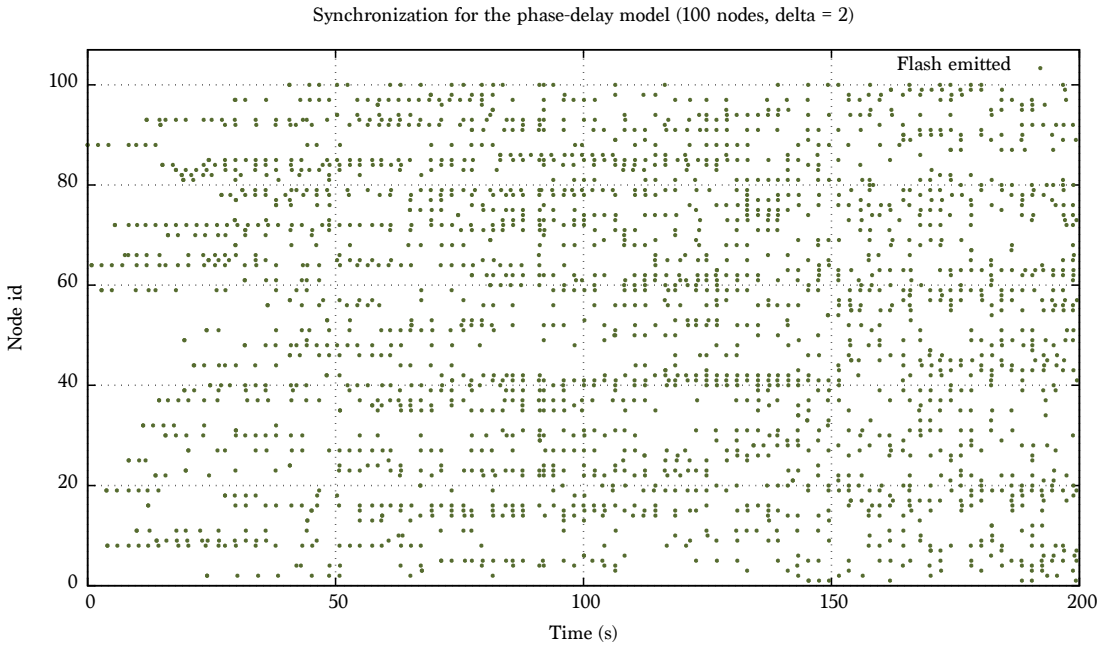

Figure 6: Phase-delay model synchronization with $\Delta = 2$

We looked at three models, namely the phase-advance and phase-delay which are very similar, and the adaptive Ermentrout model. The two first model where over-simplifying the reality and made unrealistic assumptions that can not be guaranteed. The phase-advance was flooding the network, and quickly the network would behave like a continuous big flash that would stop when some node crash. The phase-delay is a little bit better and show signs of convergence, but not for all nodes.

The adaptive Ermentrout model eliminated the unrealistic assumptions, and thus represented the reality better. We shown that under good conditions, the protocol was converging to a stable state of synchronization. This was illustrated with the emission window length getting smaller through time and the cycle lengths also converging.

However, under churn, the adaptive Ermentrout model has troubles when new nodes join the network. This may come from a bug in the implementation, because the cycle length should not be under 4.5 seconds, which is the case here.

To conclude, the adaptive Ermentrout model is a good protocol to synchronize a large-scale system in a decentralized manner. Here we only considered a network of 64 nodes, but in the reference paper, it was shown that the protocol converges quickly to a stable state for a network of 2^{16} nodes.

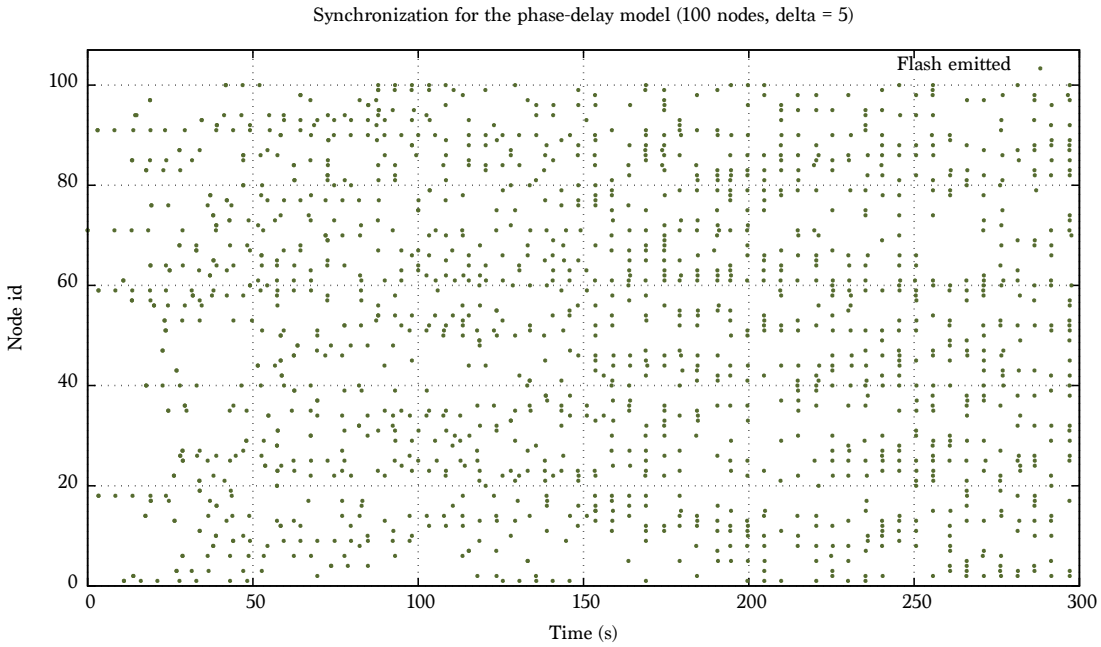

Figure 7: Phase-delay model synchronization with $\Delta = 5$

Algorithm 4.1 Pseudo-code for the adaptive Ermentrout model

Variables:
$\Delta_l,\ \Delta,\ \delta_i,\ \Delta_u \ \leftarrow\ 4.5,\ 5,\ \text{rand}(\Delta_l, \Delta_u),\ 5.5$
$\Omega_l,\ \Omega,\ \omega_i, \Omega_u \ \leftarrow\ \frac{1}{\Delta_u},\ \frac{1}{\Delta},\ \frac{1}{\delta_i},\ \frac{1}{\Delta_l}$
$\varepsilon \leftarrow 0.01$

function PROCESSFLASH()
    $\omega_i \leftarrow \omega_i + \varepsilon(\Omega - \omega) + g^+(\varphi)(\Omega_l - \omega) + g^-(\varphi)(\Omega_u - \omega)$
end function

$g^+(\varphi) \leftarrow \max\left(\frac{\sin(2\pi\varphi)}{2\pi}, 0\right)$
$g^-(\varphi) \leftarrow -\min\left(\frac{\sin(2\pi\varphi)}{2\pi}, 0\right)$

function UPDATEPHI()
    if $\varphi < 1$ **then**
        $\varphi \leftarrow \varphi + \omega_i \cdot$update_phi_period
    else
        fire event "Flash!"
        $\varphi \leftarrow 0$
        start new thread ACTIVETHREAD
    end if
end function

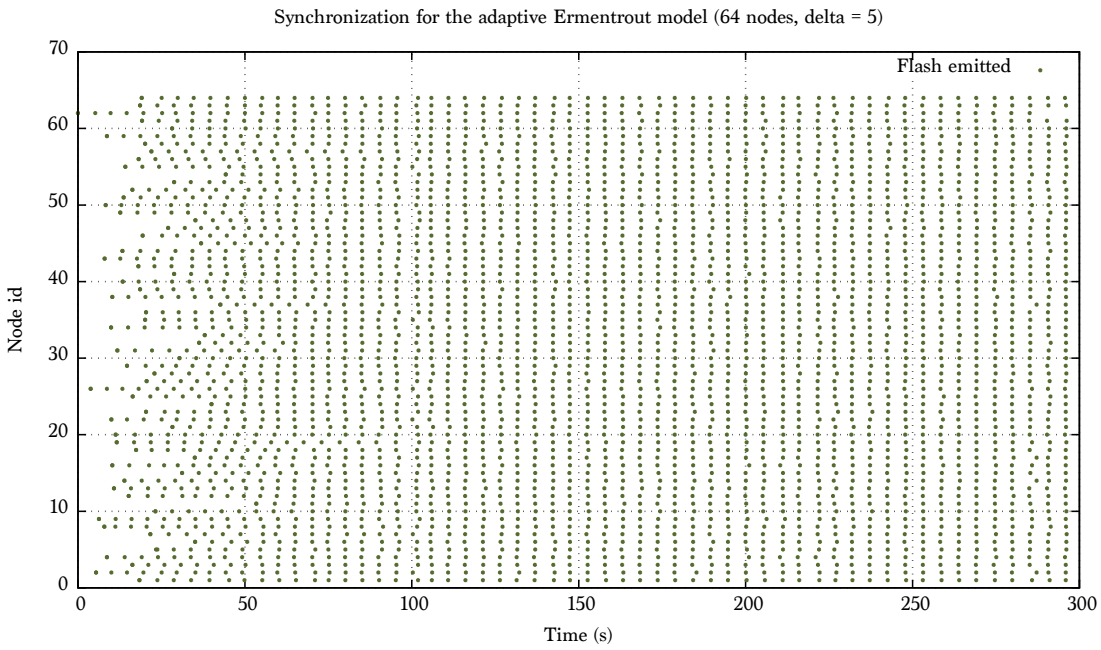

Figure 8: Adaptive Ermentrout model synchronization

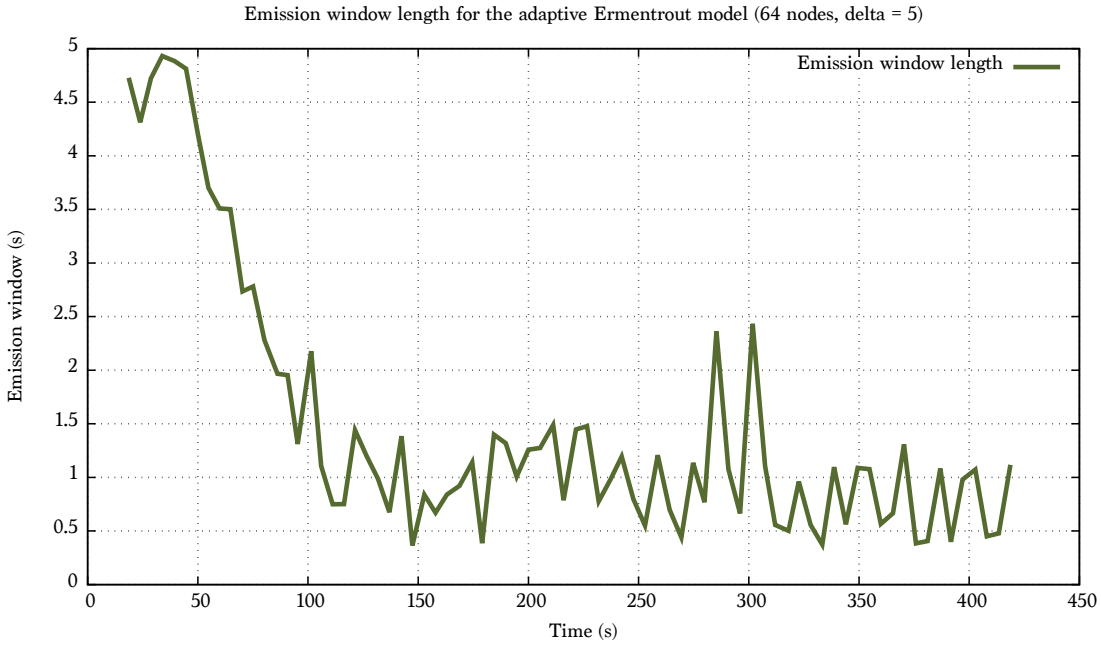

Figure 9: Adaptive Ermentrout model emission window length

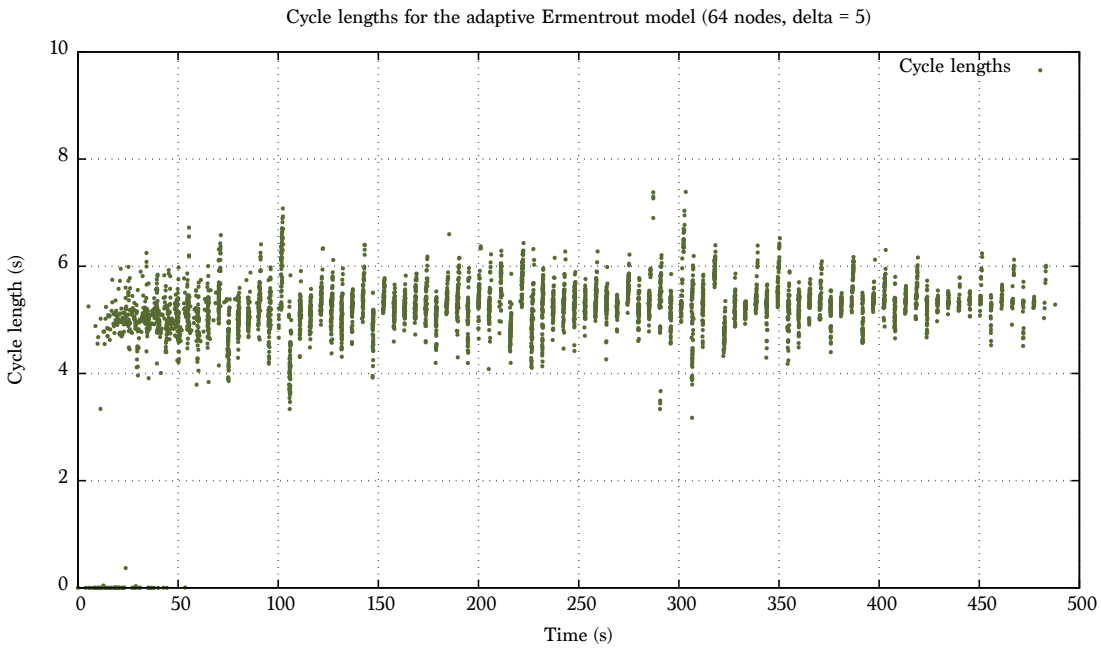

Figure 10: Adaptive Ermentrout model cycle length

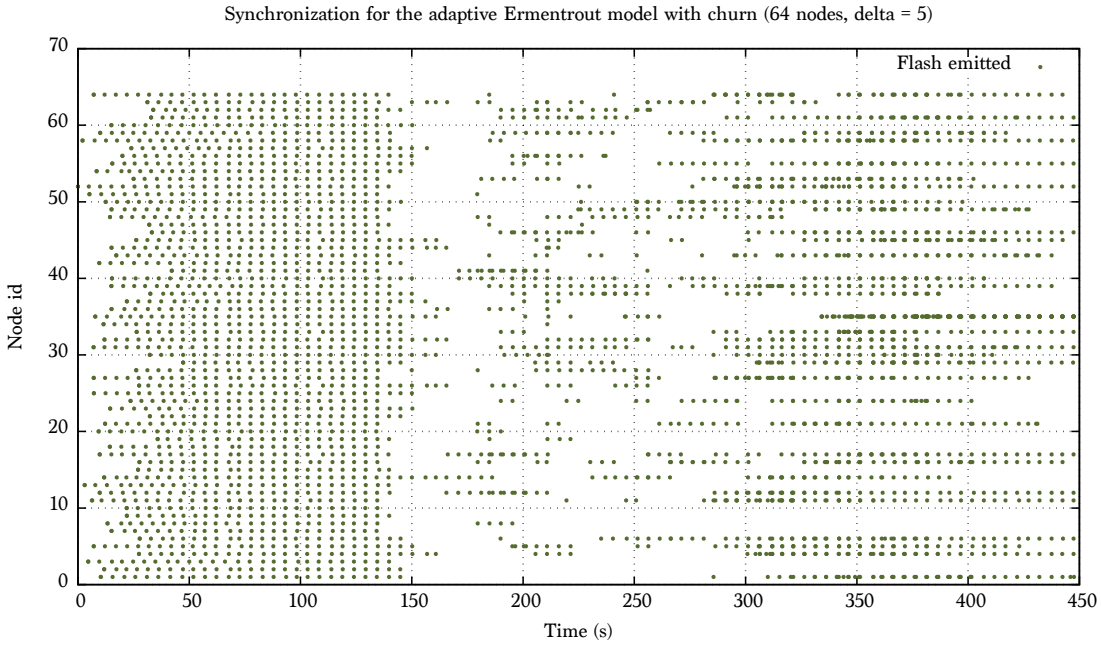

Figure 11: Adaptive Ermentrout model synchronization under churn

Figure 12: Adaptive Ermentrout model emission window length under churn

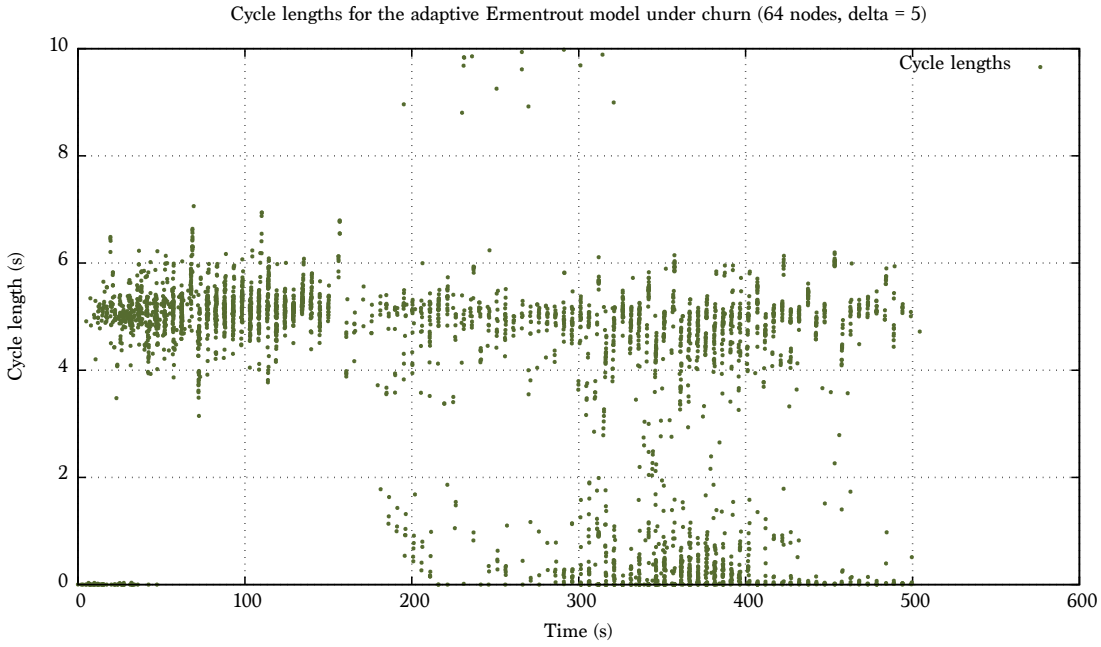

Figure 13: Adaptive Ermentrout model cycle length under churn