Breaking Links: Evaluating distributed time synchronization for musical applications using Ableton Link

Xavier Riley

Submitted for the Degree of Master of Science in

Distributed and Networked Systems

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

August 15, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name:

Date of Submission:

Signature:

Abstract

With the recent explosion of connected musical devices, the challenge of making these play "in time" with each other mounts against application developers and device manufacturers. Ableton Link[4] aims to provide robust, resilient musical synchronization using principles from distributed systems programming, in contrast to previous master/slave approaches. However the evaluation criteria for such a system are not well represented in the existing literature, with particular reference to a musical context. The following presents a system for empirical testing of the Ableton Link library using the Jepsen[6] testing framework, along with a set of criteria for evaluting similar libraries that may be developed in future.

Contents

1 Introduction

The synchronizing of events is fundamental to our perception of musical performance. Where performers are using networked connected devices, the challenge maintaining accurate synchronization incorporates the well studied problem of clock synchronization from distributed systems.

As network technologies continue to grow in usage and importance to musical performances[8], it becomes increasingly important to find common approaches to allow devices and applications to synchronize, without reliance on expensive proprietary technology or protocols which are difficult to implement and configure. The need to purchase or to understand such equipment creates an unnecessary barrier to entry, potentially impeding important contributions from musicians who lack the necessary financial or educational resources.

Music programming environments in the academic space are well catered for with regards to open source software, however applications in the consumer market have tended to lag behind some of these advances. Often either the production of music is limited to a single device, or additional devices are synchronized using specialized hardware dedicated to synchronizing frames e.g. MIDI, SMTPE - see Goltz[4] for a review of these methods.

1.1 Colocated vs. Geo-distributed

When considering the synchronization of musical programs and important distinction is whether the participants are "within earshot" of each other. If performers are distributed across a large geographic distance, such that they could not reasonable hear the audible output of their peers, then staying within the thresholds of tolerable latency starts to become impossible. This means that performers cannot react to each other in a typical way due to the effect of lag.

The position taken in this work is that geo-distributed performance is more a matter of maintaining a locally consistent ordering of events and relative timings, and as such is not under the scope of discussion here.

TODO: reference PhD on global metronome and other approaches to the above.

Focusing on colocated performance, the prime objective becomes to achieve and maintain synchrony between devices. Secondary concerns include maximum convergence time on any relevant notions of shared state, e.g. tempo, start/stop times.

1.2 About Ableton Link

Ableton Link aims to address some of these issues by supplying an open source, permissively licensed C++ library for integration with application code. As well as the popular digital audio workstation Ableton Live, implementations exist for a large number of mobile applications and for many of the popular music programming environments. In adition the following three design goals are stated[4]:

- Remove the restrictions of a typical master/client system
- Remove the requirement for initial setup
- Scale to a wide variety of music applications

It differs from existing approaches in that it does not rely on a master process to propagate timing information directly. Instead nodes will establish a session, using a reference to the start time of the oldest member of the group even if that member is no longer present.

Clock synchronization is performed using a Kalman filter[2] which adds a level of robustness to jitter introduced by the network along with a more accurate reflection of the real delay under certain conditions. This approach appears to be relatively sophisticated when compared with other music programs using basic averaging algorithms such as NTP.

Implementation is handled by application developers who are left to integrate a small API. C++ has widespread support for integration with many popular languages, making this viable for the majority of existing applications.

Setup for the end user is virtually transparent - network discovery takes place automatically on all interfaces. Clock synchronization is performed automatically for nodes joining a session and then at 60 second intervals afterwards. Simple transport commands (start, stop) and tempo changes are also propagated automatically by reliable broadcast.

Failures, drop outs and re-entry are all handled with a model of eventual consistency where last-write-wins takes effect for changes in tempo and transport state.

1.3 Evaluating synchronization for music systems

With the advent of distributed systems for music applications, there are natural questions around how these should be assessed to determine their effectiveness. Traditional criteria from distributed systems research may be useful, particularly around the bounds for timing synchronization, however musical performance doesn't have the same requirements as traditional databases, for example. A musical tempo can diverge and converge again with only minor consequences, however avoiding a double spend for a bank account requires more careful treatment.

This means that bounds for synchronization and notions of data consistency may be relaxed if doing so would benefit some other aspect, such as ease of setup/integration.

1.4 Fundamental problems of music synchronization

To characterize some of the challenges more specifically, the following list covers some of the key criteria for success:

- Getting clocks in sync [1, Chapter 6.3.2]
- Keeping them in sync (in the presence of drift or variable latency) [1, Chapter 13]
- Network bandwidth (ensuring scalability as number of devices grows)
- Fault tolerance
- Ease of setup and deployment

2 Background research

The use of networks in computer music is an active area of study, with much of the research being driven by "laptop orchestras"[11] centered around academic institutions. This has led to approaches centering around the Open Sound Control protocol[13][8][9] as the "lingua franca" of connected musical applications, although older methods include the use of MIDI, SMPTE and other standards (see [4]).

Ableton Link uses a custom network protocol but all networking is handled via private methods within the library, meaning that application developers choosing to implement it do not need to concern themselves with network level code.

2.1 Finding bounds - the limits of human perception

In order to determine some lower and upper bounds on the level of synchrony required, data gathered around musical perception will be useful. While the consensus is not complete[5], one can assume that musical performers can tolerate up to 40ms of latency between sources. This is of course dependent on the individual performer as well as other factors such as the frequency domain of the sound.

In addition to delays that impact live performers, any delay introduces the risk of comb filtering effects on sound from multiple sources. This occurs when the frequencies from once sound source reinforce or cancel out those from another sound source.

Finally, the accurate synchronization of multiple speakers is essential for the use of sound spatialization effects, such as stereo panning or surround sound.

With these in mind, the lower bound would ideally be zero (perfect synchronization) but this is unlikely in practice. Delays of 0.05ms could theoretically introduce comb filtering effects in the audible frequency range at 10kHz[7] so this may offer a more practical lower bound.

In terms of an upper bound, 40ms of delay would appear to be the upper limit in terms of the impact on the majority of listeners. Where human performers are involved, this is more likely to impact their ability to play, so a lower figure of 20ms may be more appropriate. TODO ref paper

2.2 Prior testing approaches

LANdini[9] was designed for use with a specific laptop orchestra in mind and was "tested" in rehearsal and performance. Functional testing is also described in their paper regarding reliability of message delivery and bandwidth usage.

A more rigorous testing scheme is proposed in the development of PiGMI[10] (The Raspbery Pi Global Metronome) in which metronome pulses were produced by the synchronized device at 120 pulses per minute for a duration of 30 minutes. The output is then recorded into separate channels on a sound card and later analyzed to determine offsets. In addition to this, commercial drum machines were also measured, synchronized using MIDI time clock, to add a benchmark.

This method of testing allows for excellent accuracy measuring "time at speaker" which is arguably the most realistic metric available; However, the recording, analysis methods and the reliance on physical hardware makes replication of results more challenging.

A similar result may be achieved using virtual audio inputs on a single machine e.g. using the Soundflower application on OS X, however such solutions are subject to their own sources of latency. In addition, testing within a single machine in this way would not allow for detecting drift in the system clock, relative to real time, if additional reference time sources are not also used.

3 Criteria and design

The testing approach taken in this work chooses not to measure the audio output but instead analyzes the reported times from each of the application nodes for each beat at the current tempo. This allows for simple reproduction of results and a more flexible way to test different failure modes and network conditions.

This is achieved using the Jepsen framework[6], which was originally designed to test safety guarantees under fault injection, however it is also flexible enough to test systems with less strict guarantees.

The framework starts 5 instances running a simple Ableton Link application inside Linux containers using Docker. In addition to these, a control node is started which invokes and records operations against the existing nodes. Primarily this is concerned observing stability and consistency under read and write operations against the tempo parameter of the Link session.

The Jepsen framework also allows the control node to introduce faults into the virtual network between the nodes. Under initial conditions they are all connected, however links between nodes can be cut (packets between nodes all dropped) to form different topologies. Variable or fixed latency can also be introduced. These take place via a process called a "nemesis" in the framework terminology.

The format of the test is as follows: nodes are started and allowed to join the session automatically. The control node then sequences a cycle of read, write, write operations on the tempo field with a random tempo between 20 and 220 representing the usual range of musical tempos. These operations are performed on a randomly selected node, spaced at an interval of 2 seconds apart. This continues for 60 operations totalling around 180 seconds. This figure was picked to correspond to the traditional 3 minute pop song.

Each node also logs its state at each beat (see Algorithm 1), the placement of the beat being determined by the Link protocol. These are then analysed following the test to calculate the offsets and convergence of events following changes in tempo.

This approach could be criticised as being less accurate - the processing time can introduce variations such that the status is not printed exactly on the beat - however this is more typical of the usage in a real world implementation. In measuring convergence (described below) the log analysis groups the status output from all nodes to within a 100ms window to account for any variations due to processing times. The maximum tempo used in this test is 240 beats per minute which has an inter-beat interval of 250ms, larger than the uncertainty window in use.

The alternative would be to calculate the status output at the point of the next beat. This was trialled, however changes from faster to slower tempi caused occurrences where the next beat at a slower tempo would have occurred before the current beat at a faster tempo, causing entries to go "back in time". These discontinuities in the timeline were clearly unrealistic and calculating the next beat from a given point in time would have introduced unnecessary complexity to the testing procedure.

Algorithm 1 Test procedure

```
1: loop:
2: print status();
3: sleep time-until-next-beat();
4: goto loop.
```

3.1 Consistency in a musical context

The introduction of faults and perturbations to the network can cause nodes to receive updates late, resulting in divergence for the beat markers in the session. Given that Ableton Link aims to provide eventual consistency[12] provided the network remains free of partitions these should recover eventually.

This leads to an interesting question of how to quantify the divergence and the resulting effect on the music. This work puts forward that Ableton Link and similar eventually consistent systems in future should measure "musical consistency" as the time spent in agreement relative to the length of the overall session.

Another key metric would appear to be the duration for periods of divergence. If these can be minimized the resulting adverse impact on the musical performance can be limited.

3.2 Synchronization Accuracy versus Bandwidth

In order to combat clock drift, some synchronization systems will allow a high frequency of synchronization events to ensure greater accuracy. This comes at the expense of more messages being sent over the network. For example, the LANdini project opts for 3 synchronization messages (pings) every second.

For comparison, recent work by Geng et al[3] achieves synchronization on the order of 10s of nanoseconds within datacenters, however around 5 MBits/s of bandwidth is used to achieve this result.

Given that Ableton Link targets a mass market, the use of consumer grade routers should be assumed. This makes it important to minimize the number of messages sent by the protocol to avoid overloading the hardware, while striving for the figures set out in tolerable latency above.

The synchronization protocol of Link is not documented, however from analysis of the codebase it can be seen that clock synchronization events take place when a new peer is discovered, and then at 30 second intervals thereafter.

staleMessageThreshold = 50ms measurementCount = 100 inFlight = 5

on discovery of new peer j send msg ping j

recv$_{jp}ongifnot(pong.RemoteHostTime - currentHostTime) > staleMessageThresholdreject$

while(sizeof(data) ¡ measurementCount) if inFlight ¡ inFlightLimit send msg pong$_i$

recv$_{ip}ingsend < Host, GHost, pong >$

This involves the sending of at least 50 outbound messages and receiving at least 50 inbound messages under the default settings (these are not currently configurable by a public interface). Assuming a successful run every 30 seconds with no rejected messages, this works out at 3.33 messages per second on average.

3.3 Use of Jepsen, Docker and log parsing

While Jepsen is primarily designed to exercise safety guarantees of distributed systems under partitions, the flexibility it offers allows for different kinds of tests to be performed. This work opts mainly to use Jepsen to handle the running of tests and fault injection, whereas the analysis of results takes place in a separate process which handles the parsing of log files generated during the test.

The use of Docker containers allows straightforward reproducibility across all major platforms. One possible application would be in a continuous integration (CI) testing environment so that changes to the Link codebase could be exercised against these tests automatically.

As the Link protocol favours transparent setup over configuration, there is a limited about of information available regarding the state of the session by default. For these tests, the debug logging in the Link library is enabled and these logs are then parsed following the test to produce the output in the results section below.

3.4 Topologies and latencies

As part of the stated aim of ease of setup, the Ableton Link protocol performs service discovery and message broadcast on all network interfaces to ensure that all connected nodes are able to join the same session. For example, nodes using a LAN may also see other nodes on a WiFi network provided that at least one node existed that was connected to both.

This feature introduces the prospect of topologies other than a connected network in practice. Jepsen allows different topologies to be defined and implemented during the test procedure. As Link operates using a type of reliable broadcast algorithm (retransmitting the state of the session on all interfaces when a valid state update is received) the topologies may be ranked according to the length of the maximum diameter of the network over which an update can be propagated.

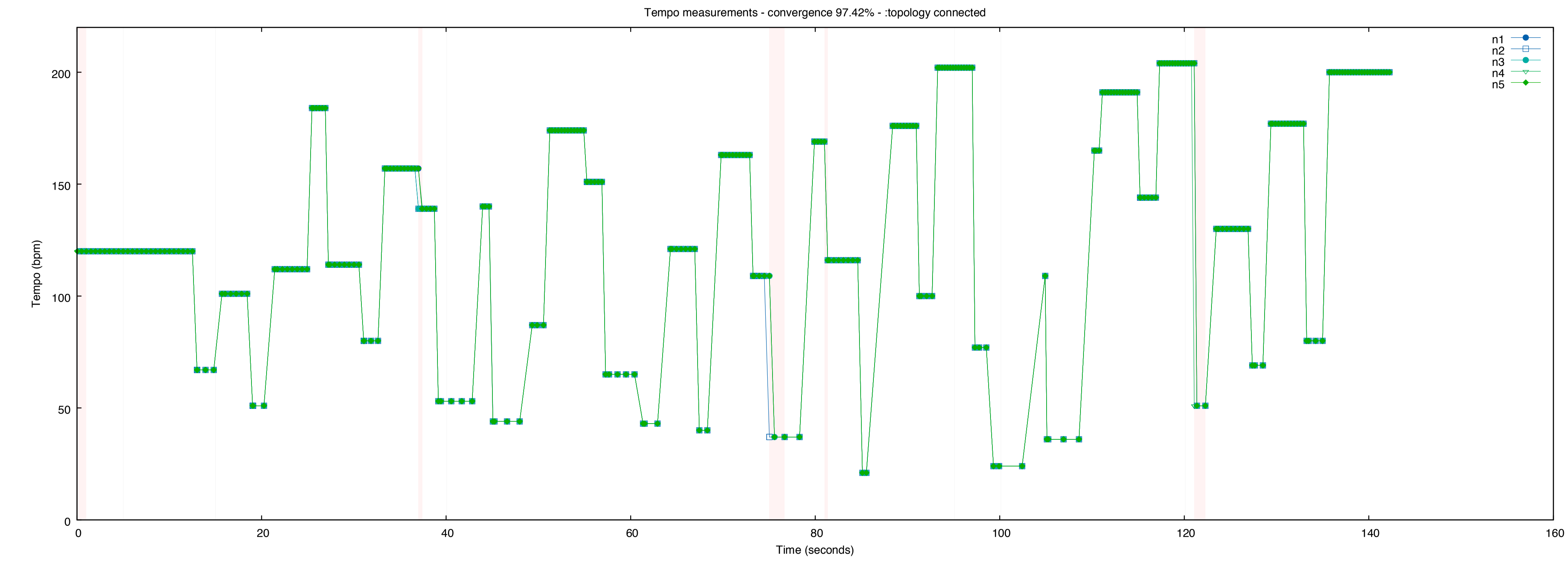
- Connected
- Bridge
- Line

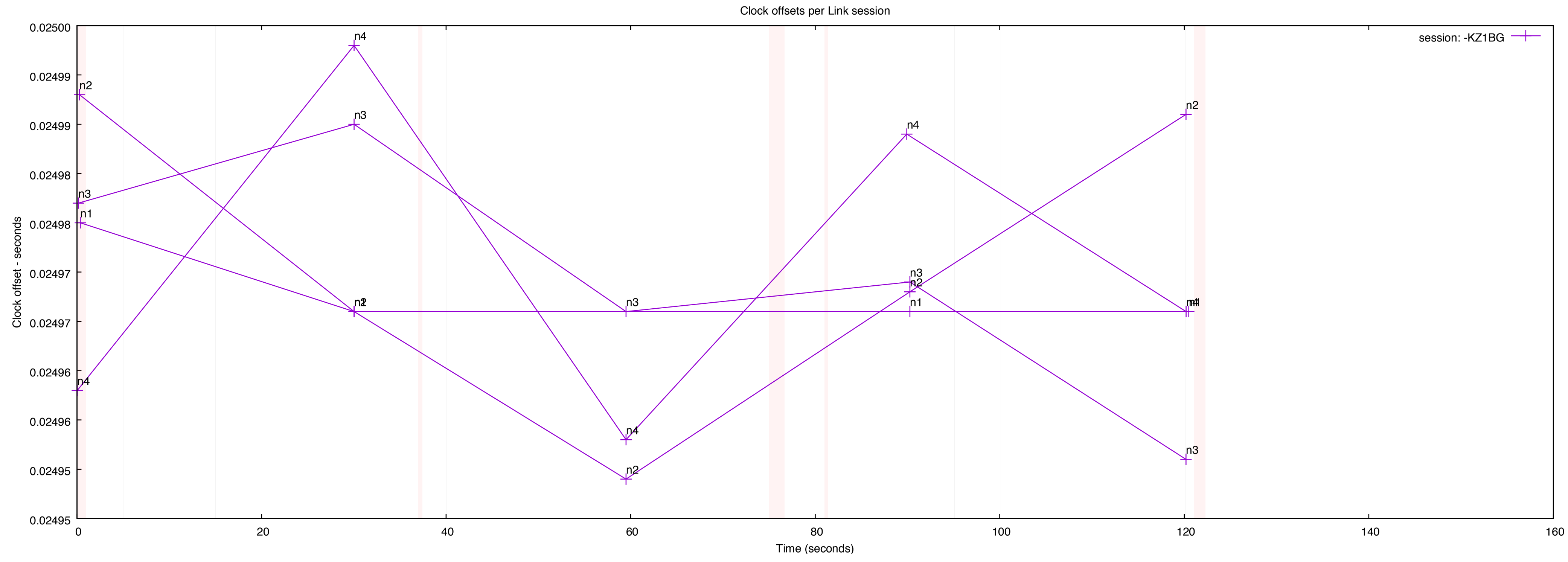
Another test condition concerns the performance of the protocol under small, constant network delays. In the tests below this is defined as 50ms delays, which for clock synchronization messages results in a round trip time delta of +100ms.

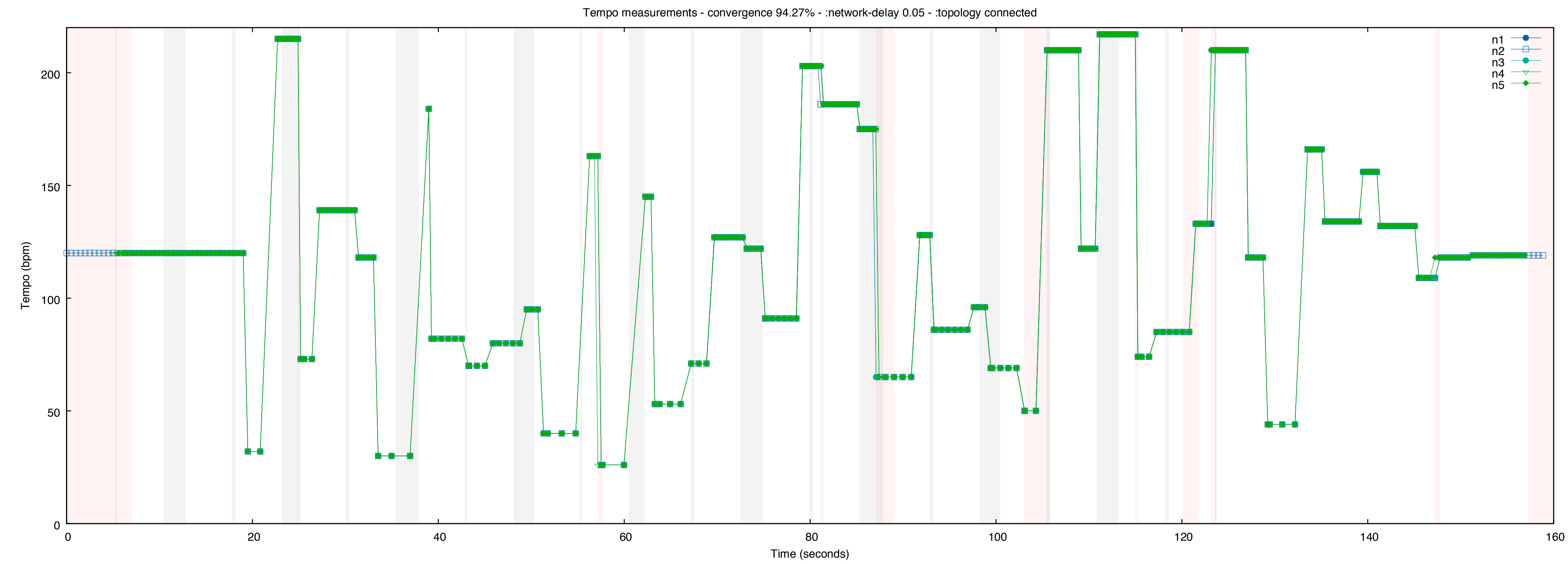
Finally the protocol is exercised in the presence of larger delays (500ms - 1s RTT). The timings for these are included on the charts below.

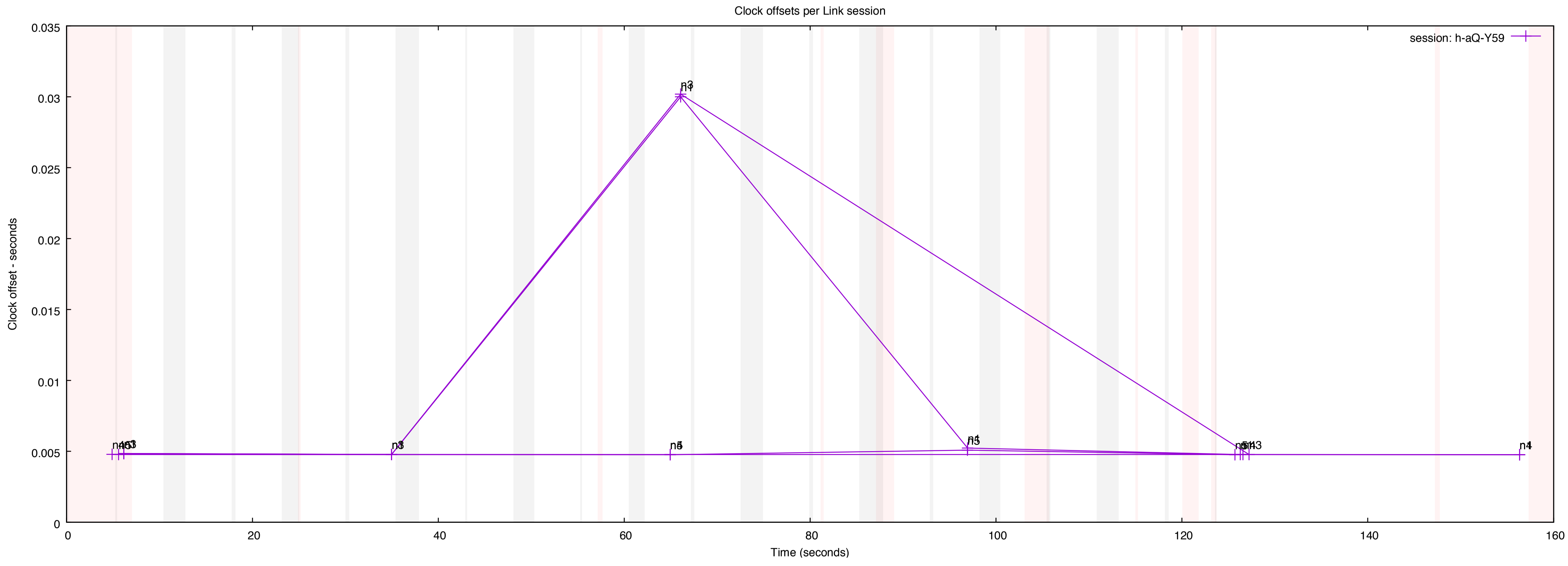
4 Experimental results

Connected, 0ms delay Connected, 50ms delay Connected, 500ms delay Bridge, 0ms delay Bridge, 50ms delay Bridge, 500ms delay Line, 0ms delay Line, 50ms delay Line, 500ms delay

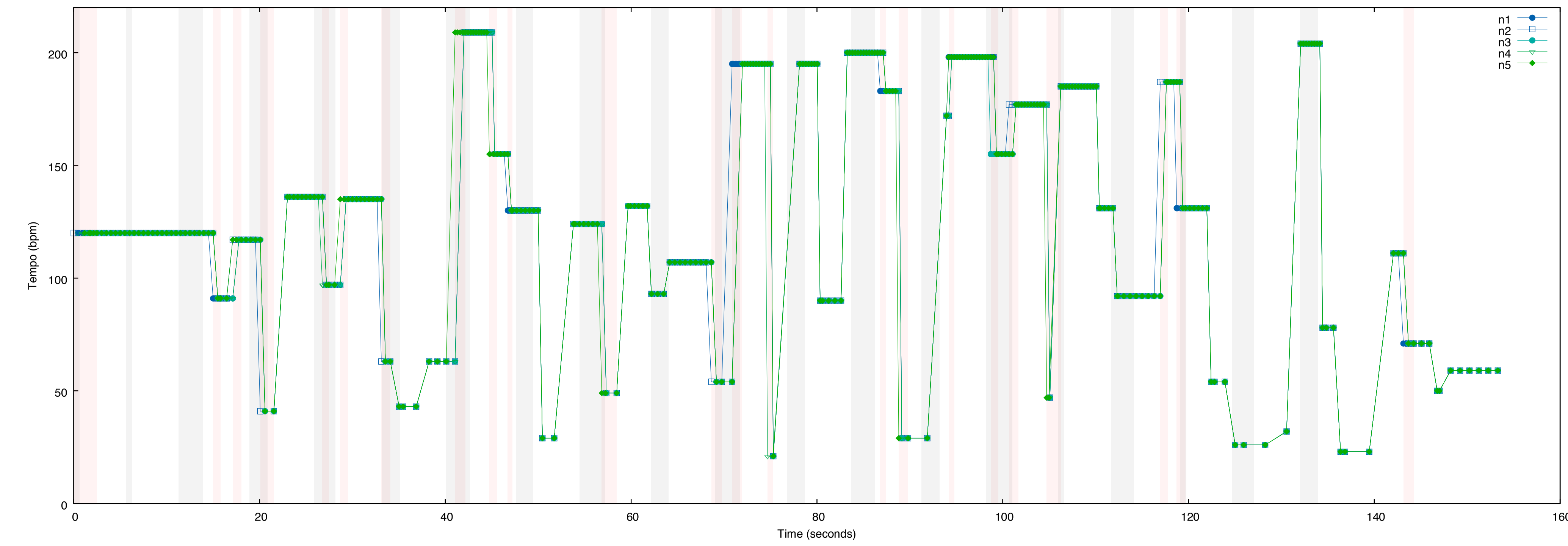

Tempo measurements - convergence 97.42% - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
0
50
100
150
200
0
20
40
60
80
100
120
140
160


Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -KZ1BG
0.02495
0.02495
0.02496
0.02496
0.02497
0.02497
0.02498
0.02498
0.02499
0.02499
0.02500
n1
n2
n3
n4

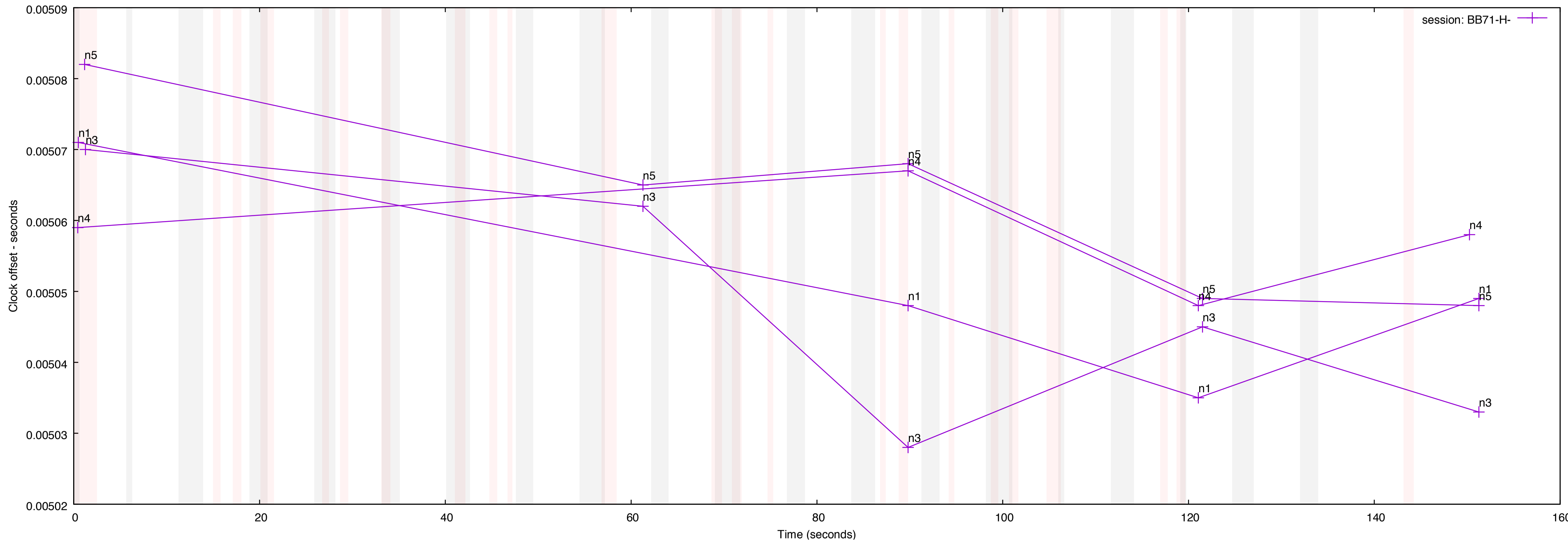

Tempo measurements - convergence 94.27% - :network-delay 0.05 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
0
50
100
150
200
0
20
40
60
80
100
120
140
160

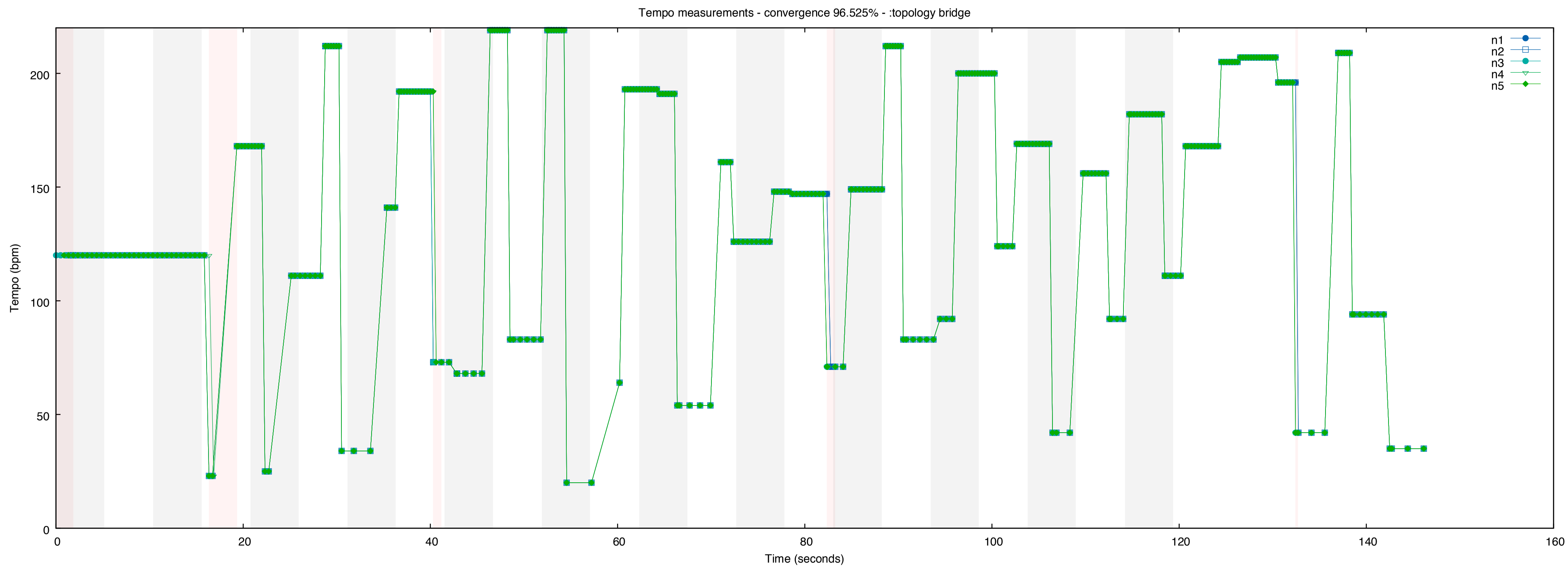

Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: h-aQ-Y59
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0
20
40
60
80
100
120
140
160

Tempo measurements - convergence 86.166% - :network-delay 0.5 - :topology connected

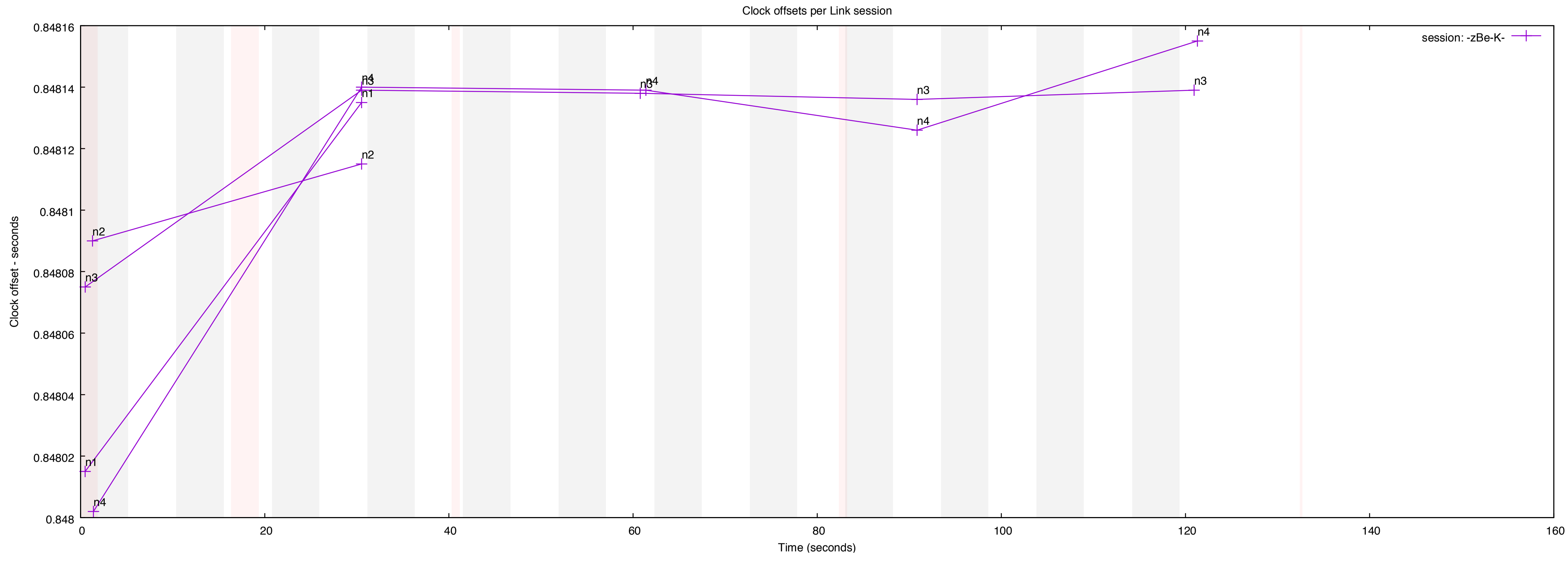


Clock offsets per Link session

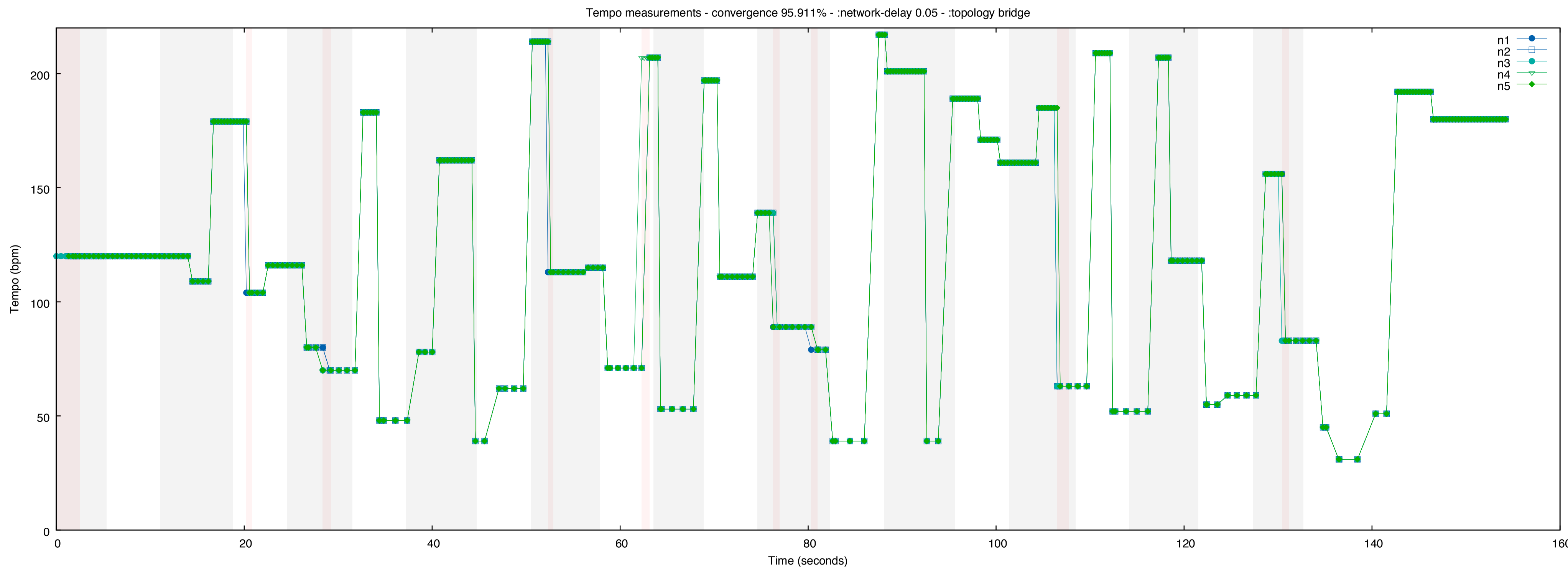


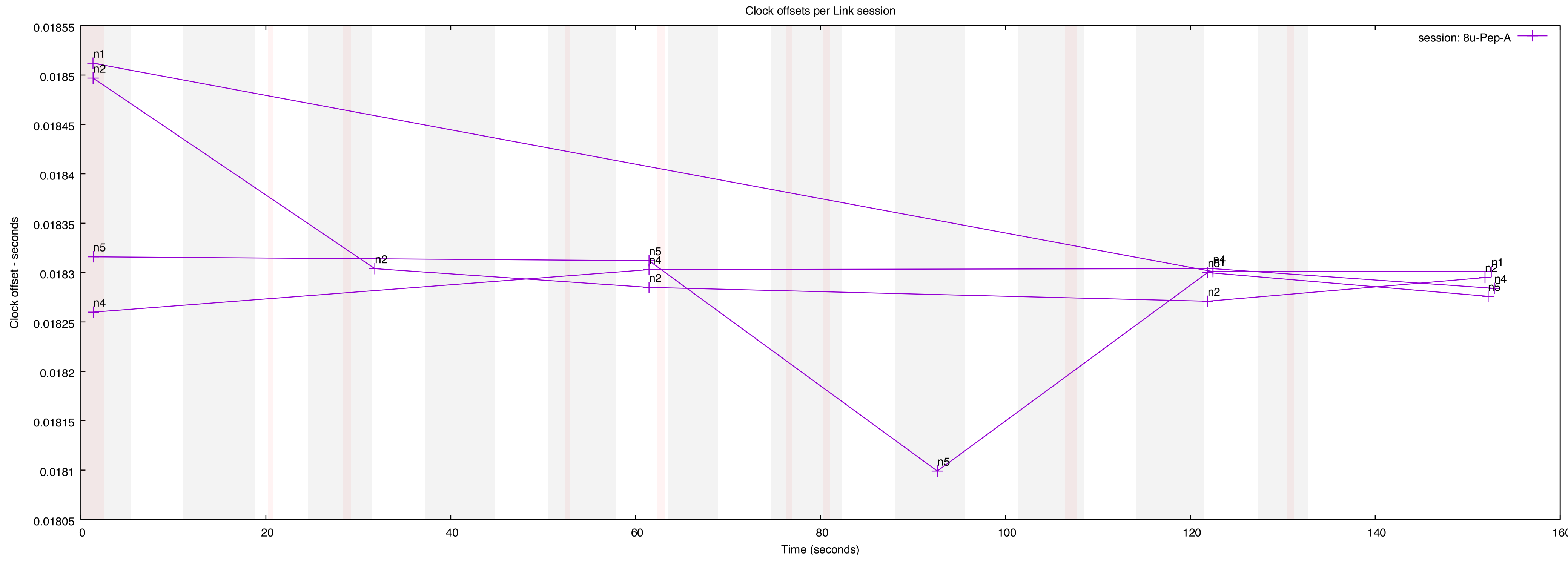


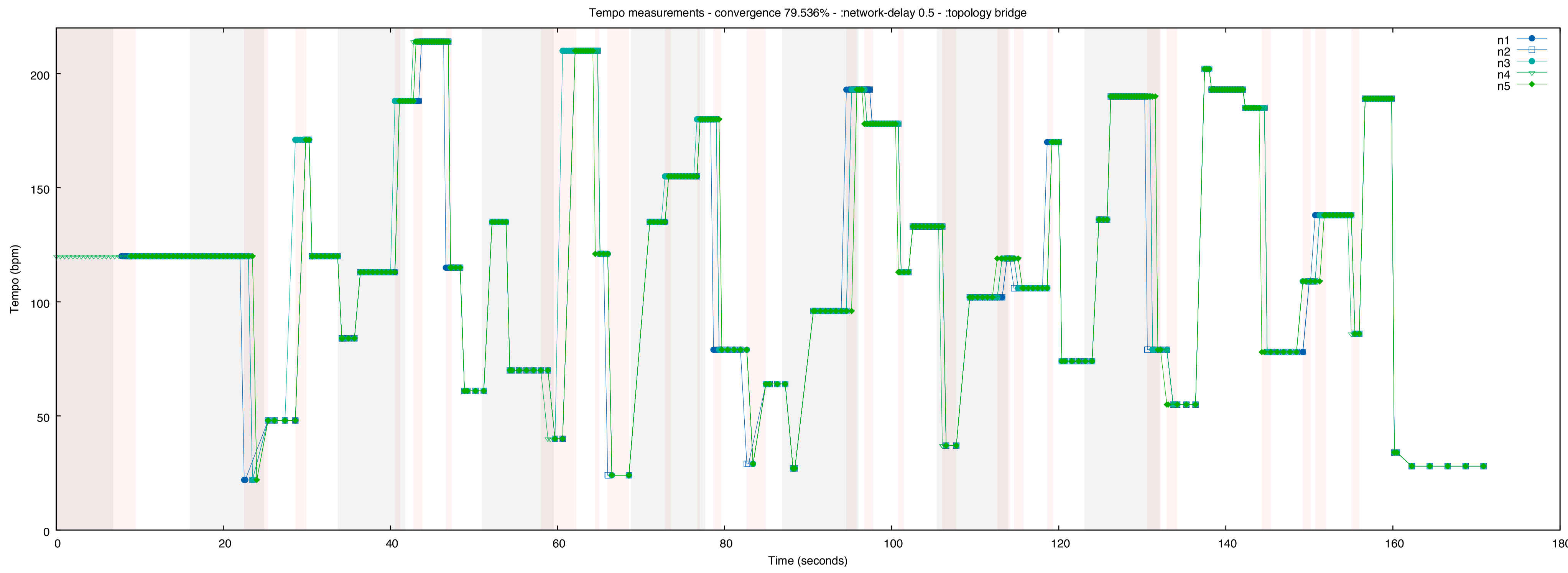
Tempo measurements - convergence 96.525% - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
0
50
100
150
200
0
20
40
60
80
100
120
140
160

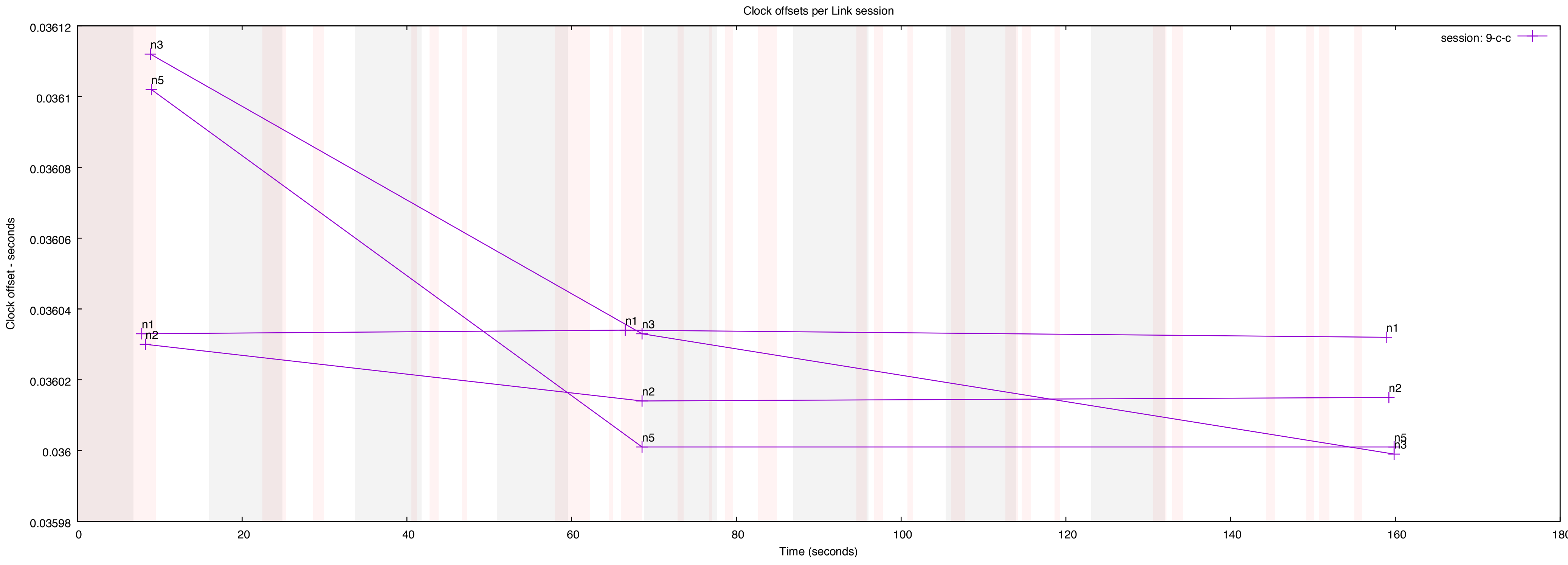


Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -zBe-K-
0.848
0.84802
0.84804
0.84806
0.84808
0.8481
0.84812
0.84814
0.84816
n1
n2
n3
n4
0
20
40
60
80
100
120
140
160

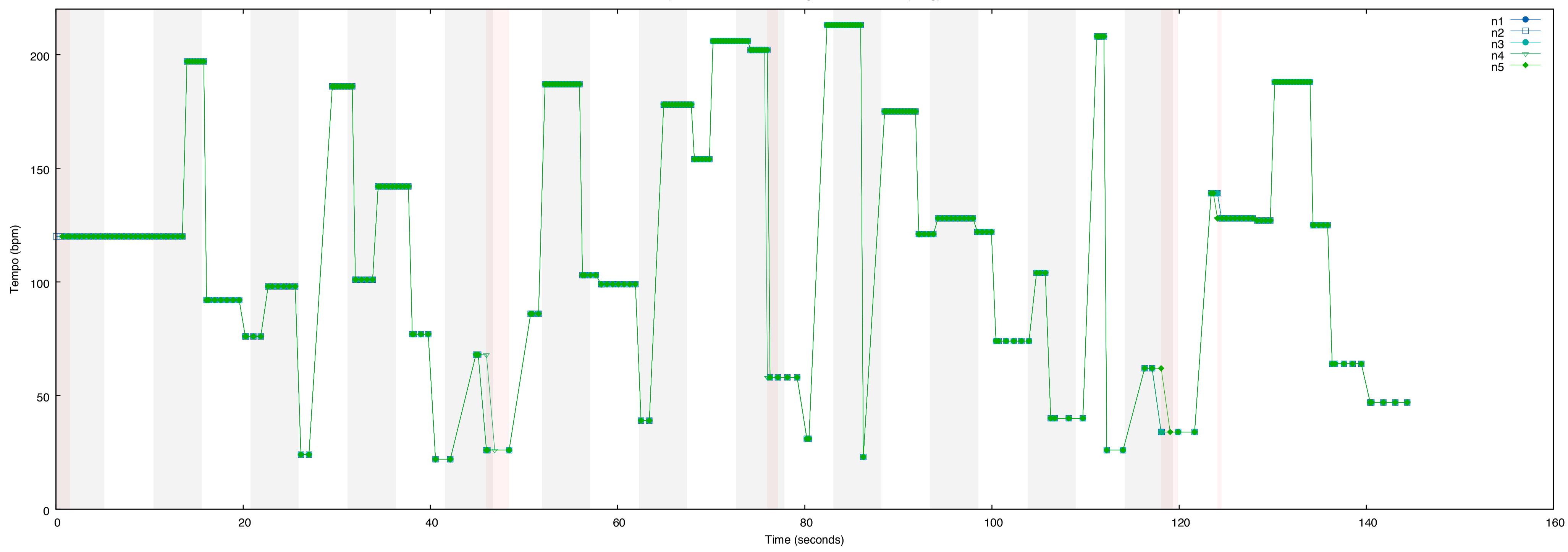

Tempo measurements - convergence 95.911% - :network-delay 0.05 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
0
50
100
150
200
0
20
40
60
80
100
120
140
160


Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: 8u-Pep-A
n1
n2
n3
n4
n5
0.01805
0.0181
0.01815
0.0182
0.01825
0.0183
0.01835
0.0184
0.01845
0.0185
0.01855
0
20
40
60
80
100
120
140
160

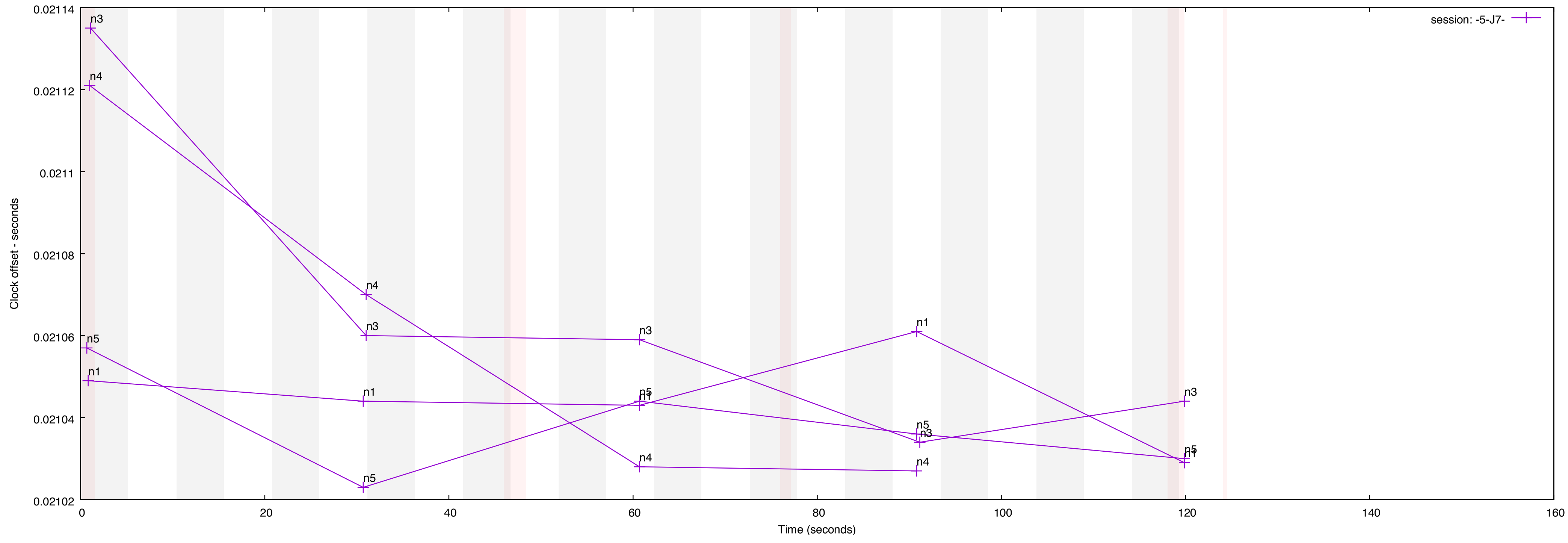

Tempo measurements - convergence 79.536% - :network-delay 0.5 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
0
50
100
150
200
0
20
40
60
80
100
120
140
160
180

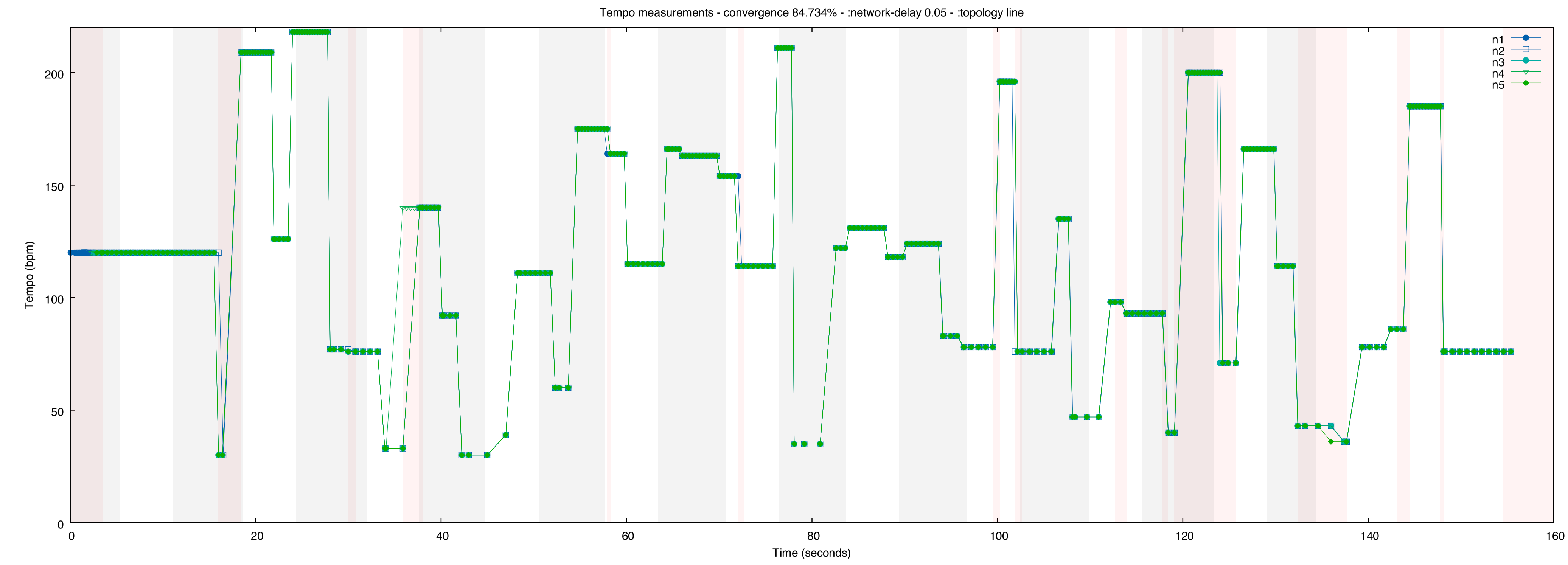

Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: 9-c-c
0.03612
0.0361
0.03608
0.03606
0.03604
0.03602
0.036
0.03598
n1
n2
n3
n5
0
20
40
60
80
100
120
140
160
180

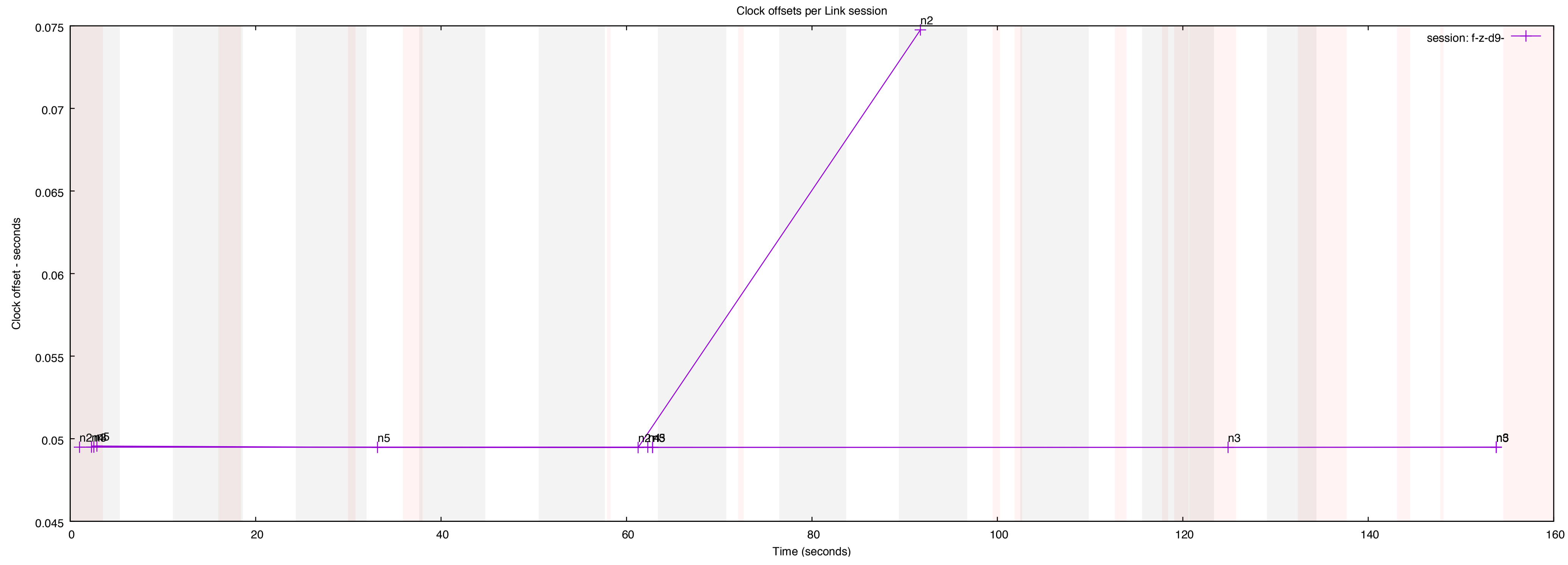
Tempo measurements - convergence 95.953% - :topology line



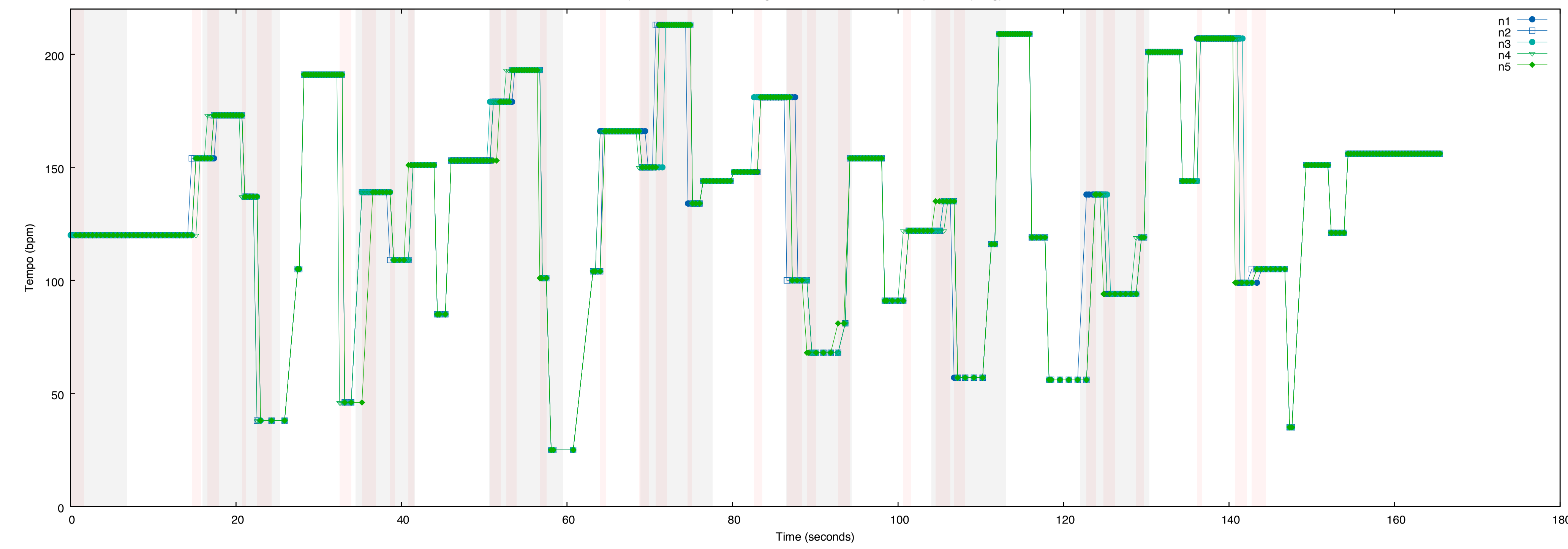
Clock offsets per Link session



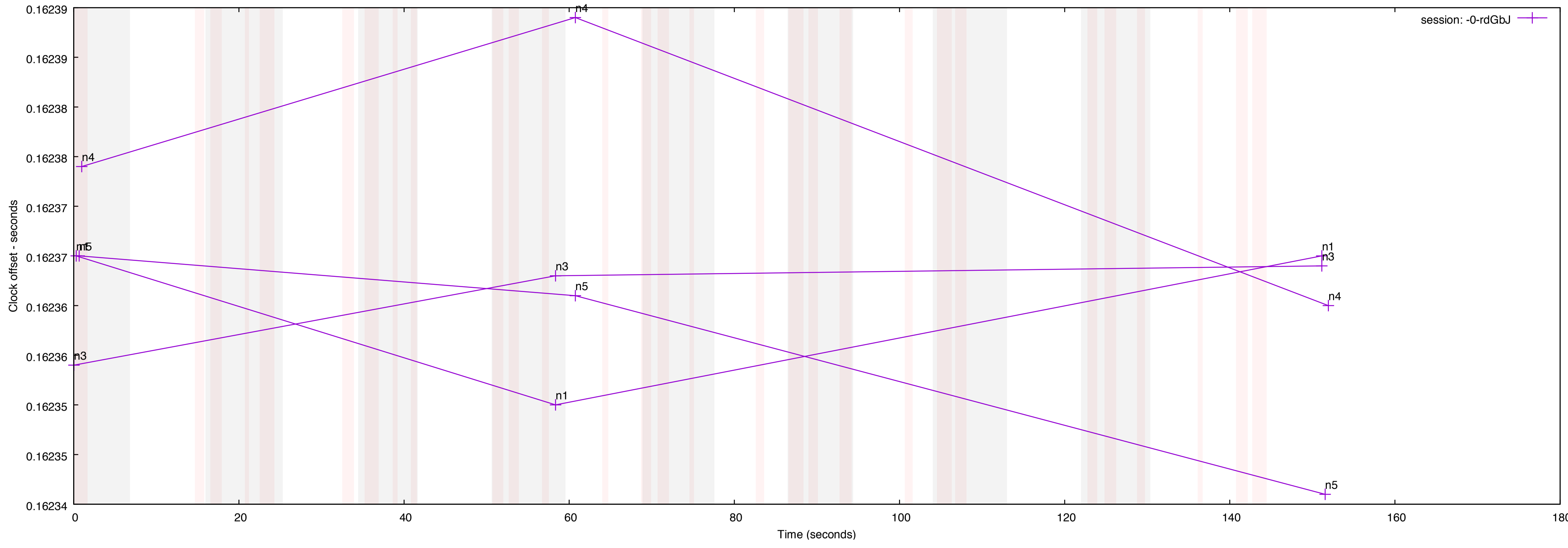

Tempo measurements - convergence 84.734% - :network-delay 0.05 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5


Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: f-z-d9-
n2
n5
n3

Tempo measurements - convergence 79.921% - :network-delay 0.5 - :topology line



Clock offsets per Link session



5 Analysis

Resilient to different network conditions. Usually converges. Musical consistency has linear? falloff.

Talk about indeterminate divergence (missed statuses). Difficulty of measurements from a library with no configuration and little in the way of callbacks.

5.1 Effects of delay distributions on clock synchronization

Delay distributions and Kalman filtering - surprising results. Expected clock sync to track delays but apparently not.

Bandwidth measurements - difficult for partitions - maybe set partitions up first to see

5.2 Identifying and triggering invalid states

Ping/pong measurement has hardcoded cap of 50ms for RTT Assuming RTT / 2 for latency calculation, this allows for max deviation of +/-25ms either side of leader Anyway, the offset can vary by delta

Node A send msgA tempo(120), t = 100 clock synchronization event - delta goes from 25ms to 0 send msgB tempo(140), t = 99.075 msgB is set locally (timestamp not checked for local updates)

Node B recv msgA - newer than last tempo - set tempo = 120 recv msgB - reject (older when timestamps are compared) broadcast state

Node A recv state 120 with beat origin 100 - update? or not?

all neighbours behave the same, therefore 120 will be accepted and converged on however this seems sub-optimal.

If recv msgA is dropped/lost when received by Node C, then Node C will be tempo = 140. If Node C retransmits with it's own beat origin (arbitrarily higher) then an inconsistent state is reached.

6 Further work

Generalize the work to other solutions.

Create CI testing for the Link library.

Suggest improvements to Link - Google nanosecond precision paper. Potential uncertainty window around synchronization events. Configuration/visibility of session information.

Integration into Sonic Pi. Windows compatibility.

7 Conclusions

References

[1] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[2] Aggelos Bletsas. Evaluation of kalman filtering for network time keeping. *ieee transactions on ultrasonics, ferroelectrics, and frequency control*, 52(9):1452–1460, 2005.

[3] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, 2018. USENIX Association.

[4] Florian Goltz. Ableton link–a technology to synchronize music software. In *Linux Audio Conference 2018*, page 39, 2018.

[5] Waldo Greeff. *The influence of perception latency on the quality of musical performance during a simulated delay scenario.* PhD thesis, University of Pretoria, 2016.

[6] Kingsbury, Kyle. Jepsen - a framework for distributed systems verification, with fault injection, 2018. [Online; accessed 6-August-2018].

[7] Michael Lester and Jon Boley. The effects of latency on live sound monitoring. In *Audio Engineering Society Convention 123*. Audio Engineering Society, 2007.

[8] Sebastian OH Madgwick, Thomas J Mitchell, Carlos Barreto, and Adrian Freed. Simple synchronisation for open sound control. In *Volume 2015*, page 218:225. International Computer Music Association, 2015.

[9] Jascha Narveson and Dan Trueman. Landini: a networking utility for wireless lan-based laptop ensembles. In *Proc. SMC Sound, Music and Computing Conference*, 2013.

[10] Reid Oda and Rebecca Fiebrink. The global metronome: Absolute tempo sync for networked musical performance. In *Proceedings of*

the International Conference on New Interfaces for Musical Expression, volume 16 of *2220-4806*, pages 26–31, Brisbane, Australia, 2016. Queensland Conservatorium Griffith University.

[11] Dan Trueman. Why a laptop orchestra? *Organised Sound*, 12(2):171–179, 2007.

[12] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[13] Matthew Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

A Self assessment

B Running the tests

C Configuring Jepsen to support other systems and tests

D Code listings

E Professional issues

F Acknowledgements

Open source code Colleagues at Heroku Salesforce for funding Gregory and staff at RHUL Em