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 18, 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 Where other published and unpublished source materials have been used, these have been acknowledged.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[7] aims to provide robust 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[11] testing framework, along with a set of criteria for evaluating 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[14], 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[7] for a review of these methods.

1.1 Co-located 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 rather than accurate synchronization, and as such is not under the scope of discussion here.

Other works have taken a different view, attempting to offer creative solutions to overcome the inherent latency with wide-area communication. In Oda[17] these include the use of GPS timing to provide a globally consistent time server, along with predictive instruments that attempt to send messages to remote clients to synthesize a performer's actions ahead of time in order to overcome latency. While interesting, in their current state these

are unlikely to be generalizable to consumer software.

Focusing on co-located 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 addition the following three design goals are stated[7]:

- 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 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[3].

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 30 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 [2, Chapter 6.3.2]
- Keeping them in sync (in the presence of drift or variable latency) [2, 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"[19] centred around academic institutions. This has led to approaches centring around the Open Sound Control protocol[22][14][15] as the "lingua franca" of connected musical applications, although older methods include the use of MIDI, SMPTE and other standards (see [7]).

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[8], 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[13] 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[4].

2.2 Prior testing approaches

LANdini[15] 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[16] (The Raspberry 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 analysed 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 analyses 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[11], 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 five 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 place-

ment 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. Ableton Link aims to provide eventual consistency[20] provided the network remains free of partitions so these should recover in time.

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[6] 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.

TODO: this algorithm

Algorithm 2 Link clock synchronization

```
    staleMessageThreshold ← 50ms
2:  measurementCount ← 100
    inFlightLimit ← 5
4:  procedure SyncProcedure
    on discovery of new peer j:
6:  send msg ping j
```

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 via 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 re-

sults 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 48ms delays on all nodes except the leader for reasons outlined below.

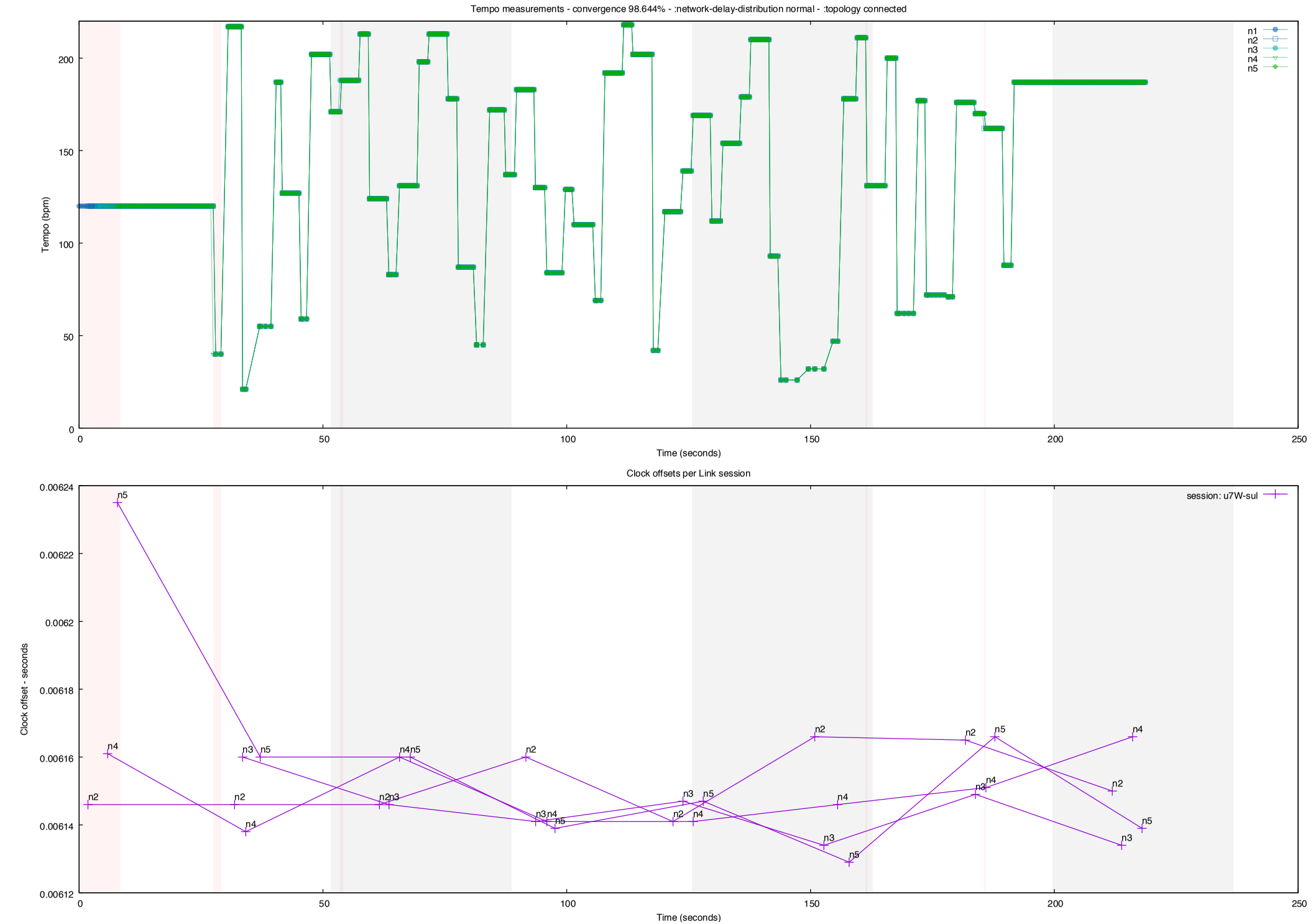
The protocol is also exercised in the presence of larger delays (500ms). The timings for these are included on the charts below.

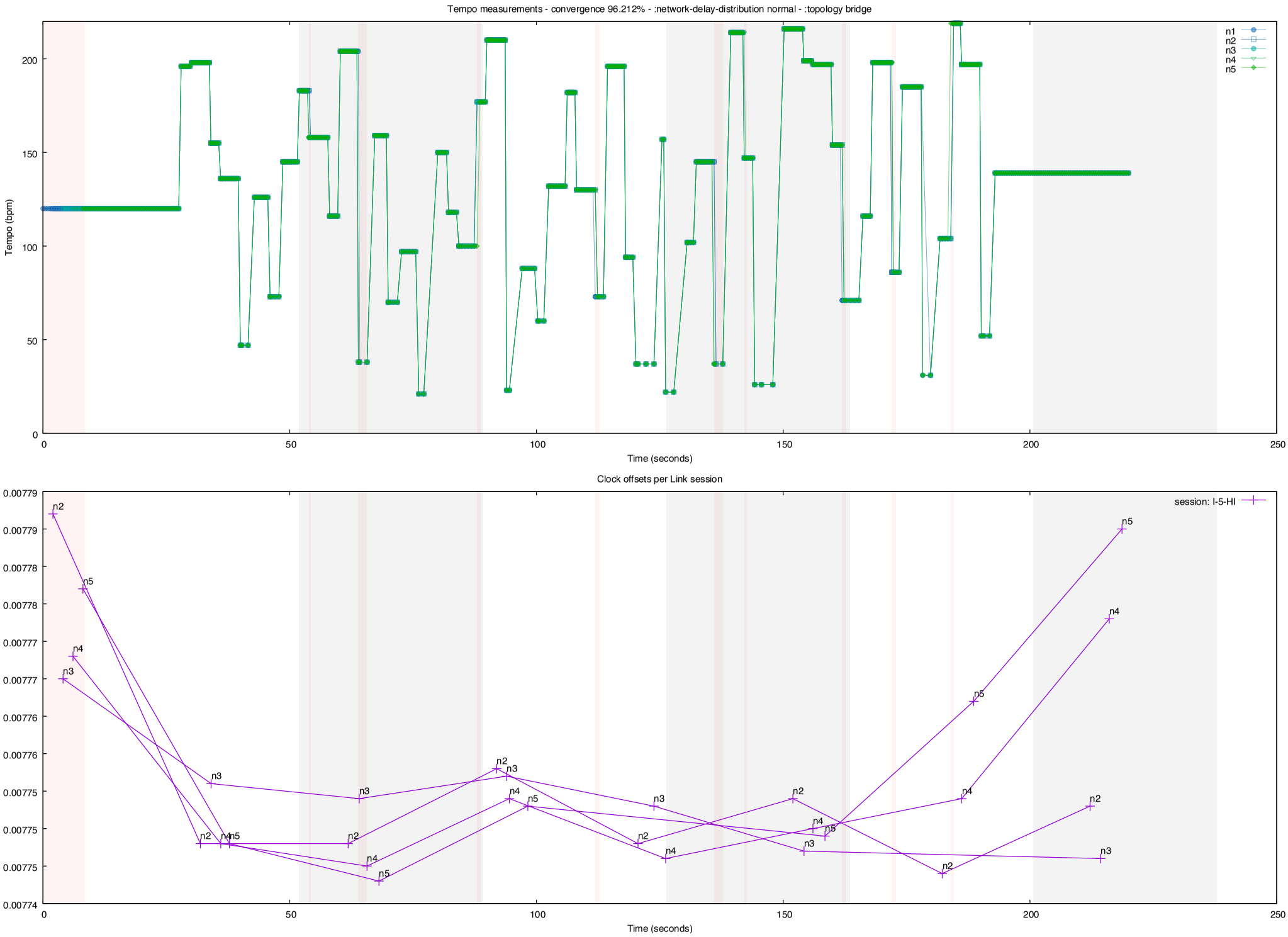
Finally, differing distributions of delays are examined as follows:

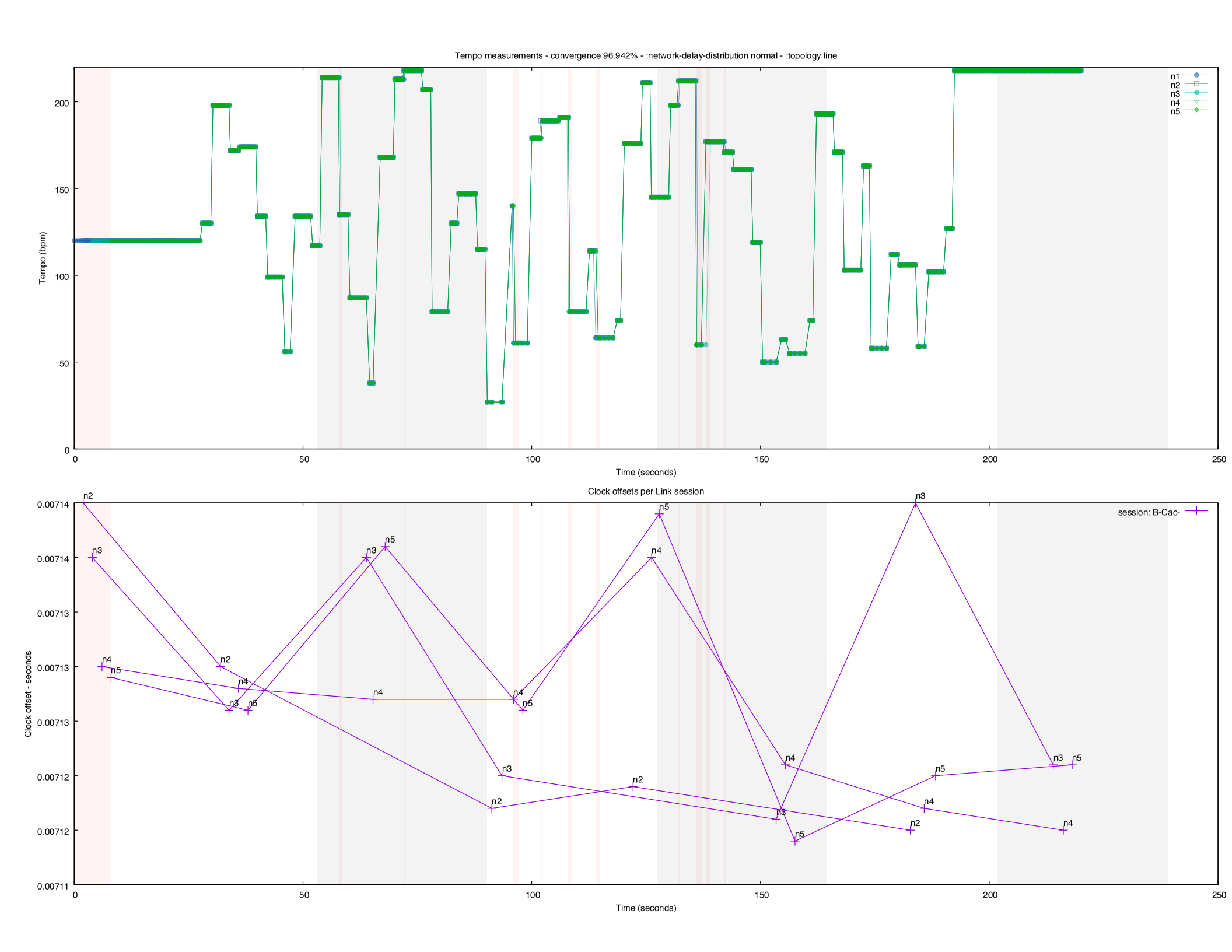
- Constant delay

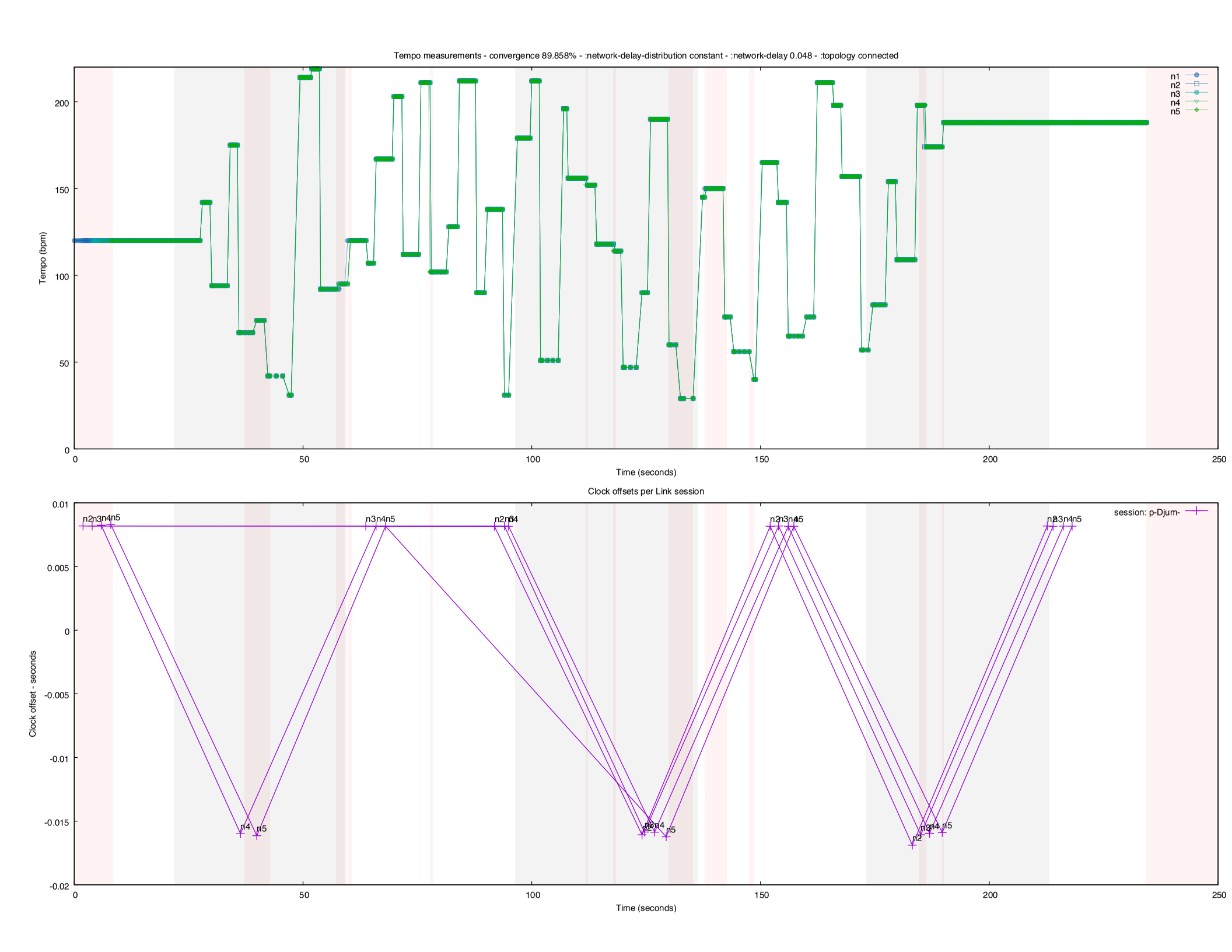
- Normal distribution
- Pareto distribution

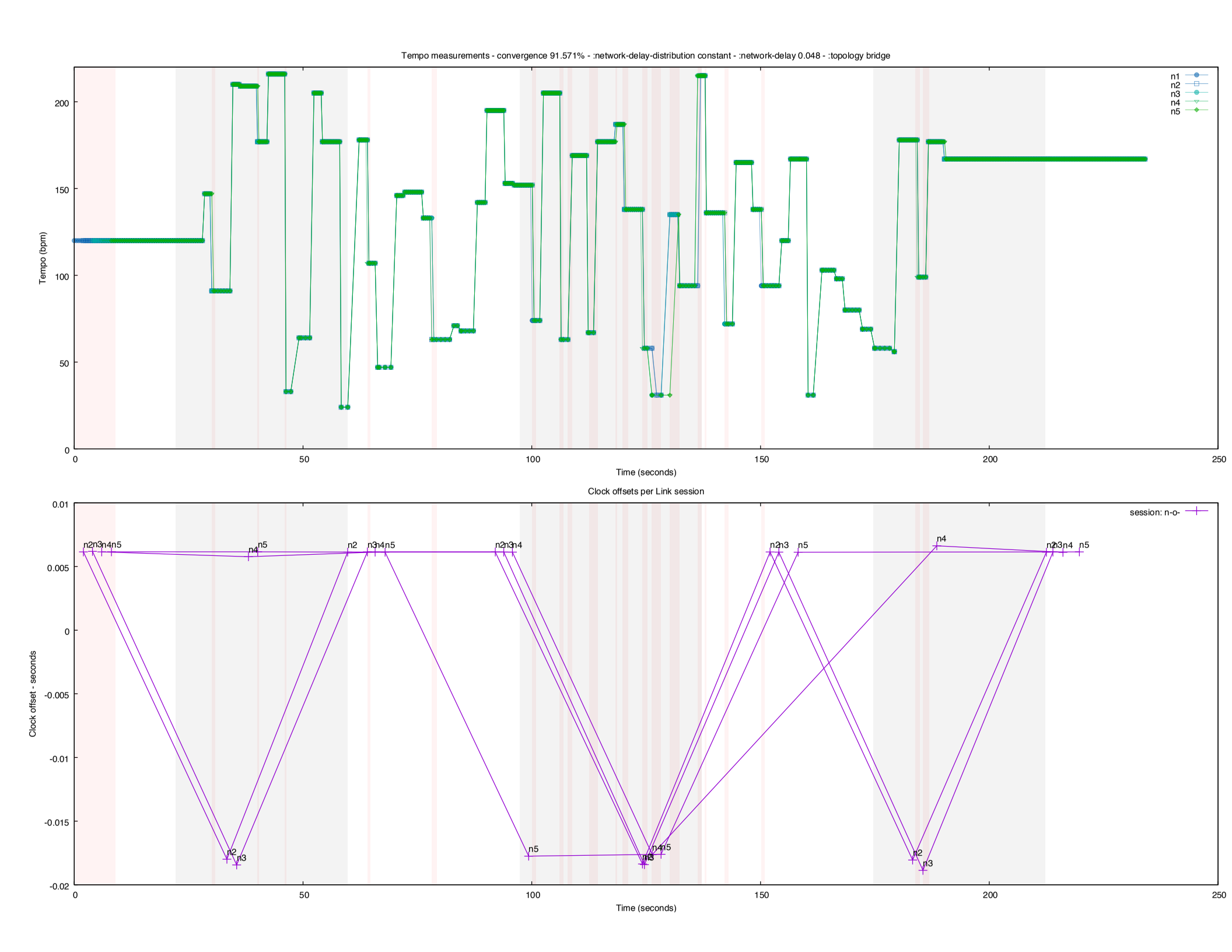
4 Experimental results

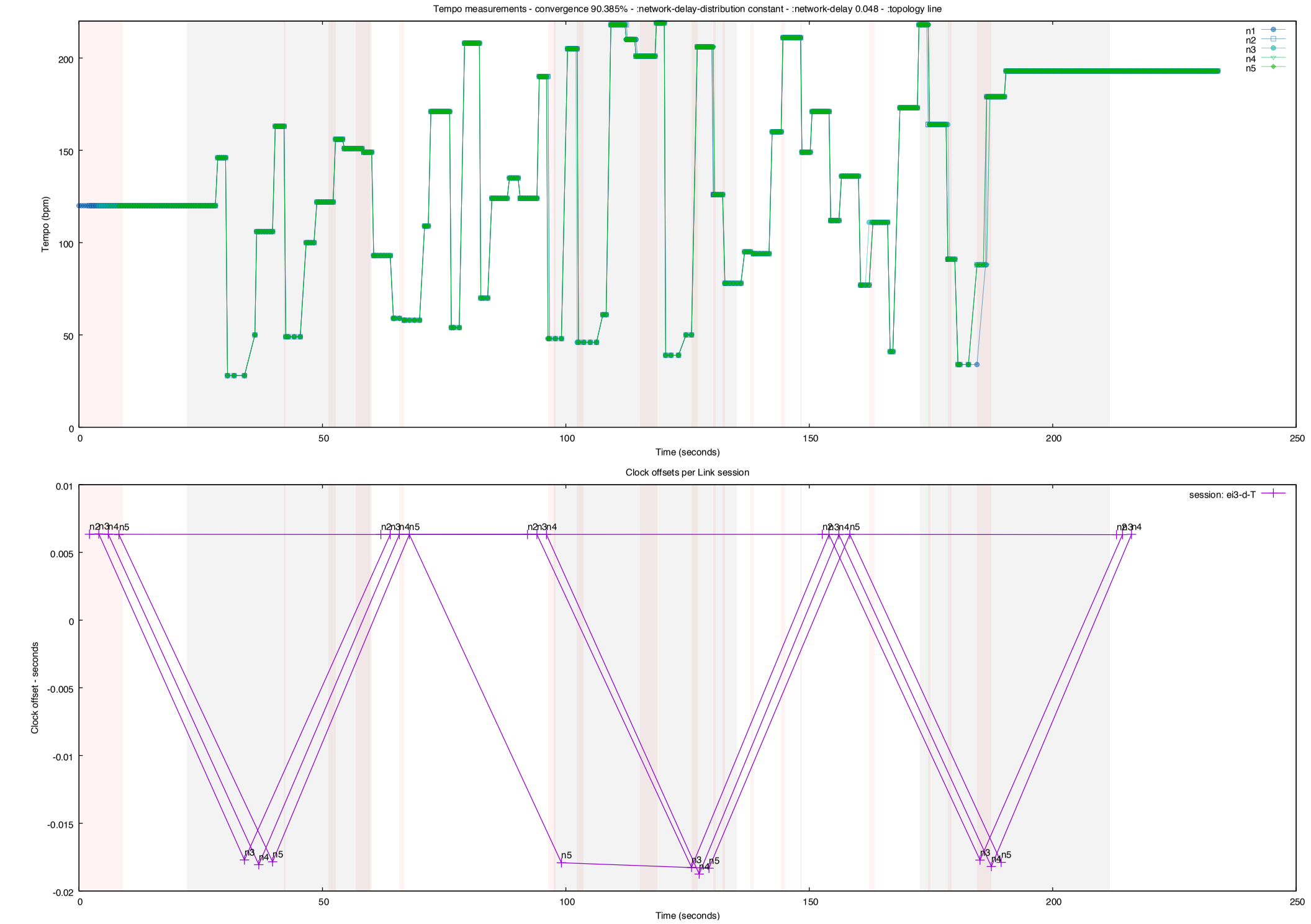

Tempo measurements - convergence 98.644% - :network-delay-distribution normal - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: u7W-sul

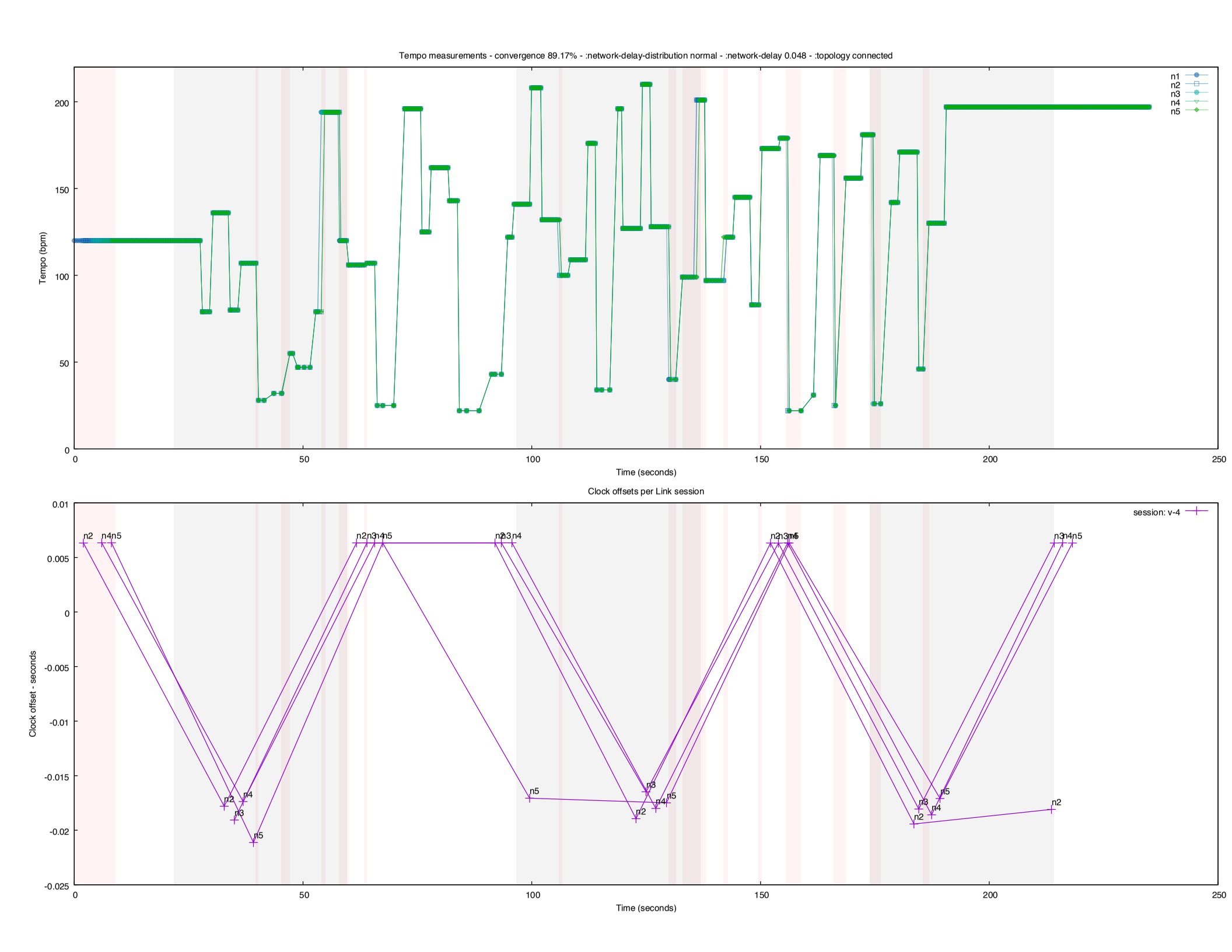

Tempo measurements - convergence 96.212% - :network-delay-distribution normal - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: I-5-HI

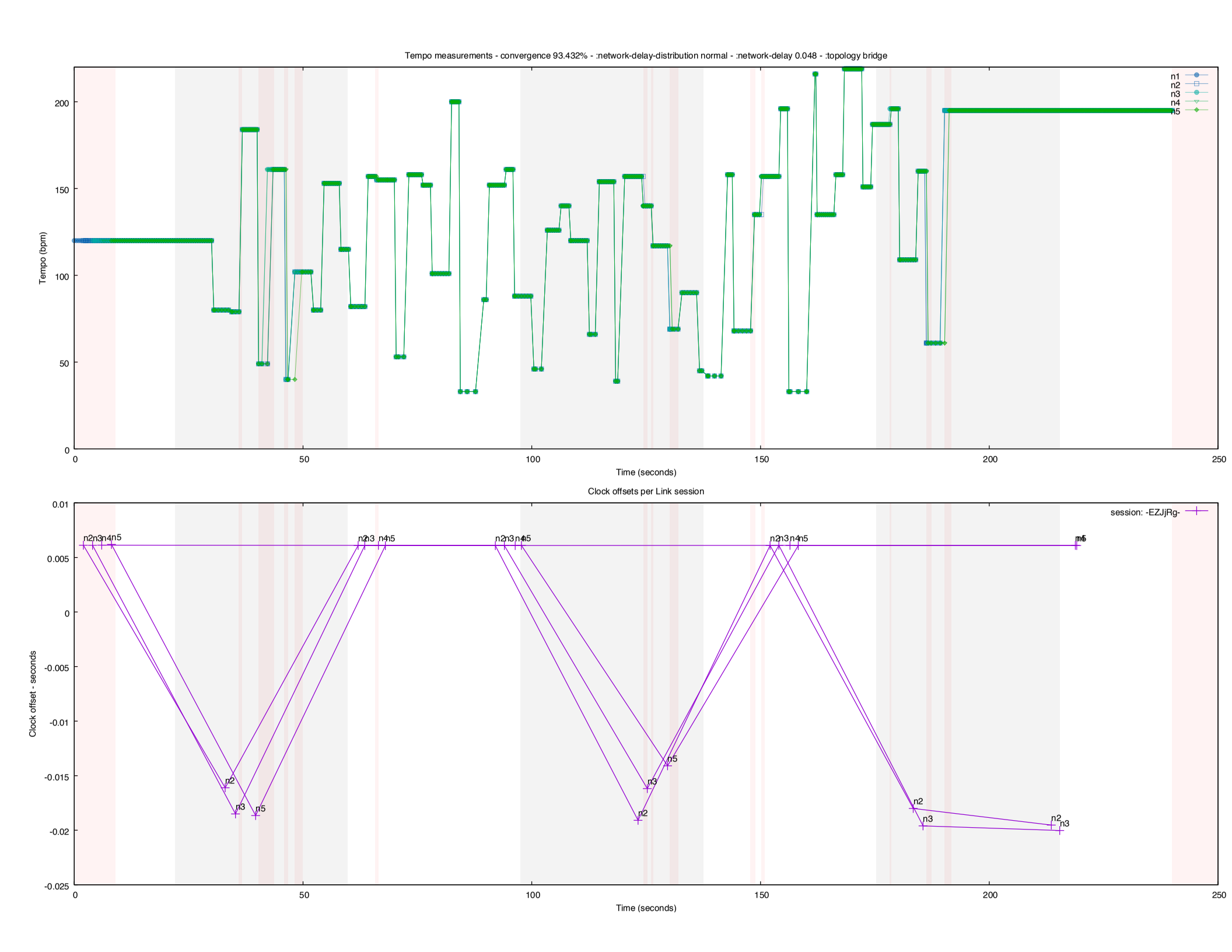

Tempo measurements - convergence 96.942% - :network-delay-distribution normal - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
session: B-Cac-

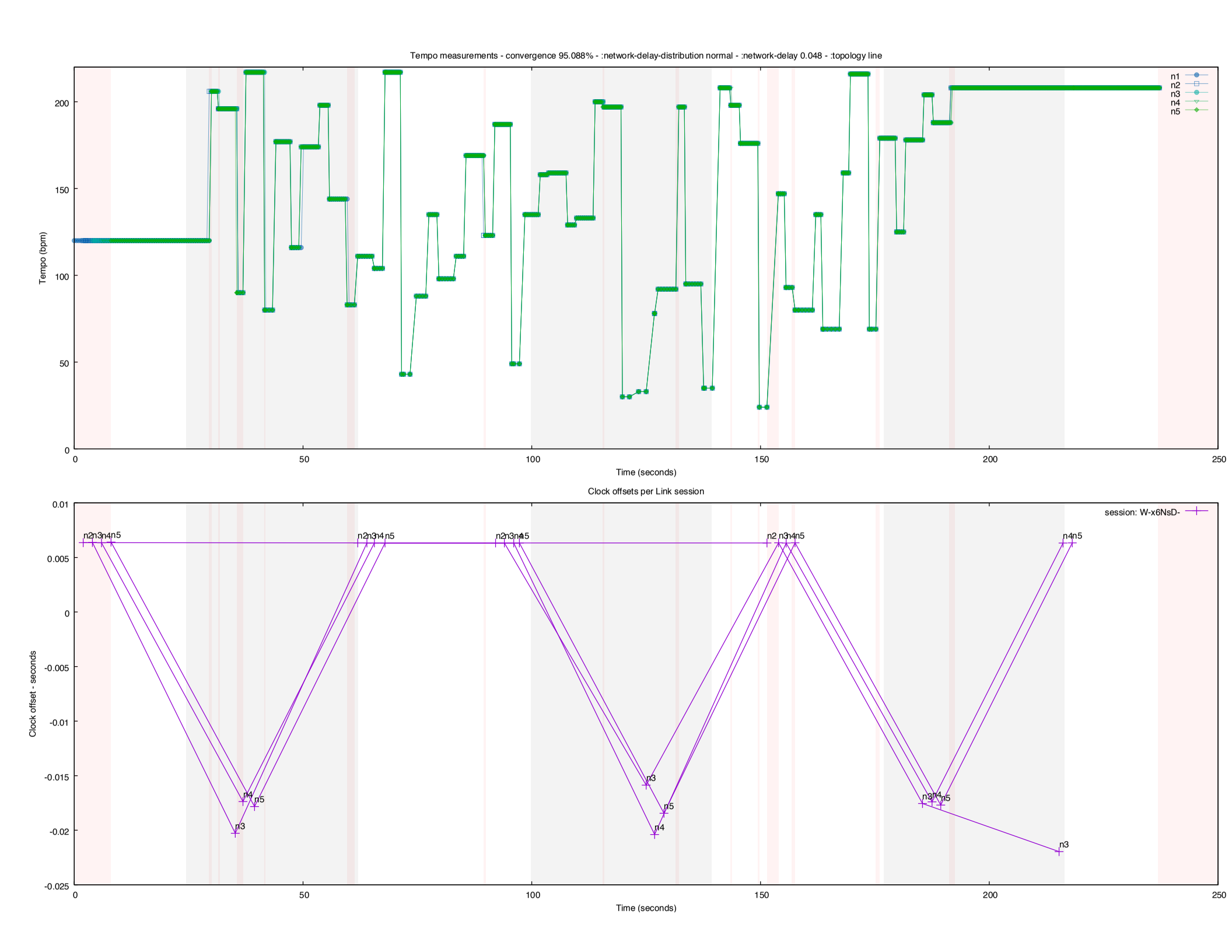

Tempo measurements - convergence 89.858% - :network-delay-distribution constant - :network-delay 0.048 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: p-Djum-

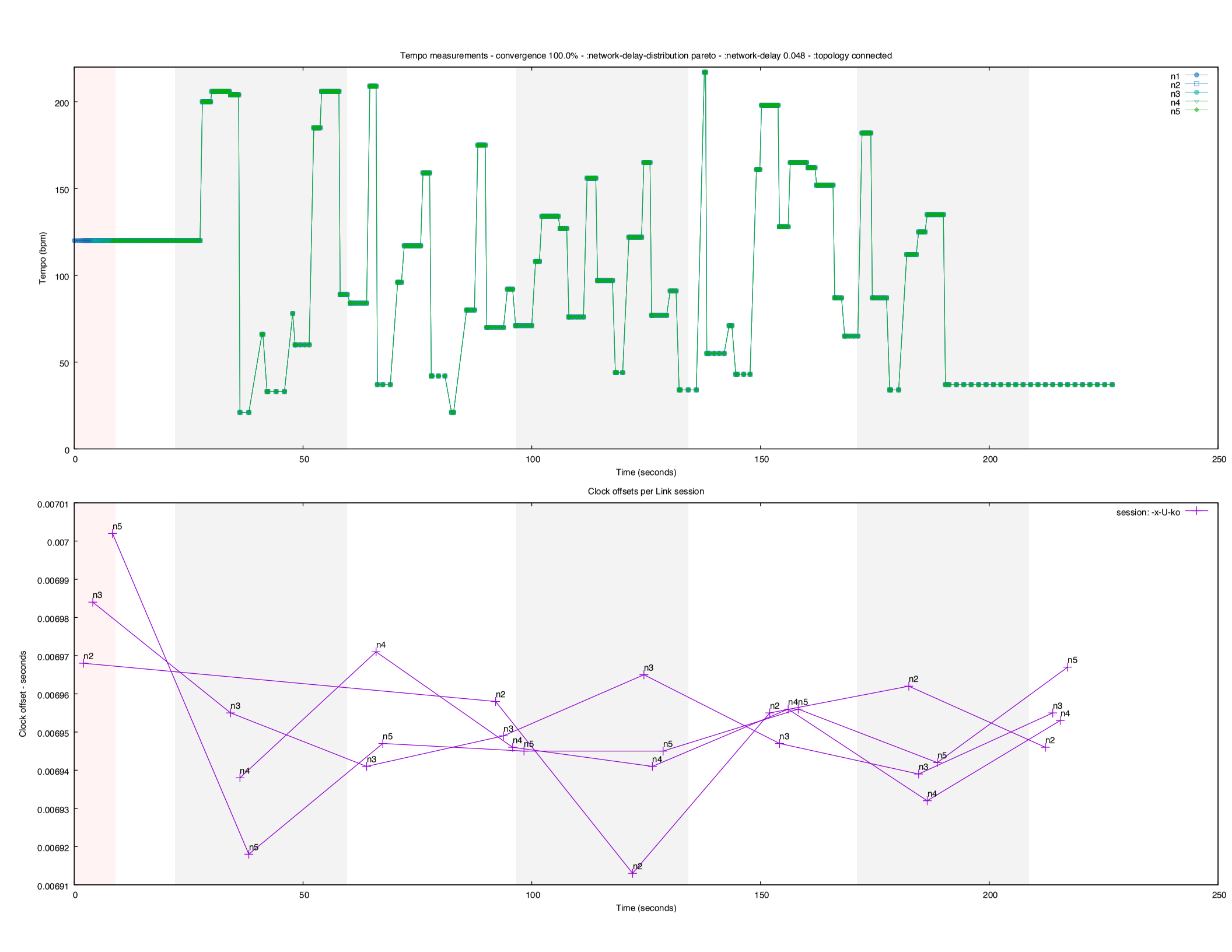

Tempo measurements - convergence 91.571% - :network-delay-distribution constant - :network-delay 0.048 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: n-o-

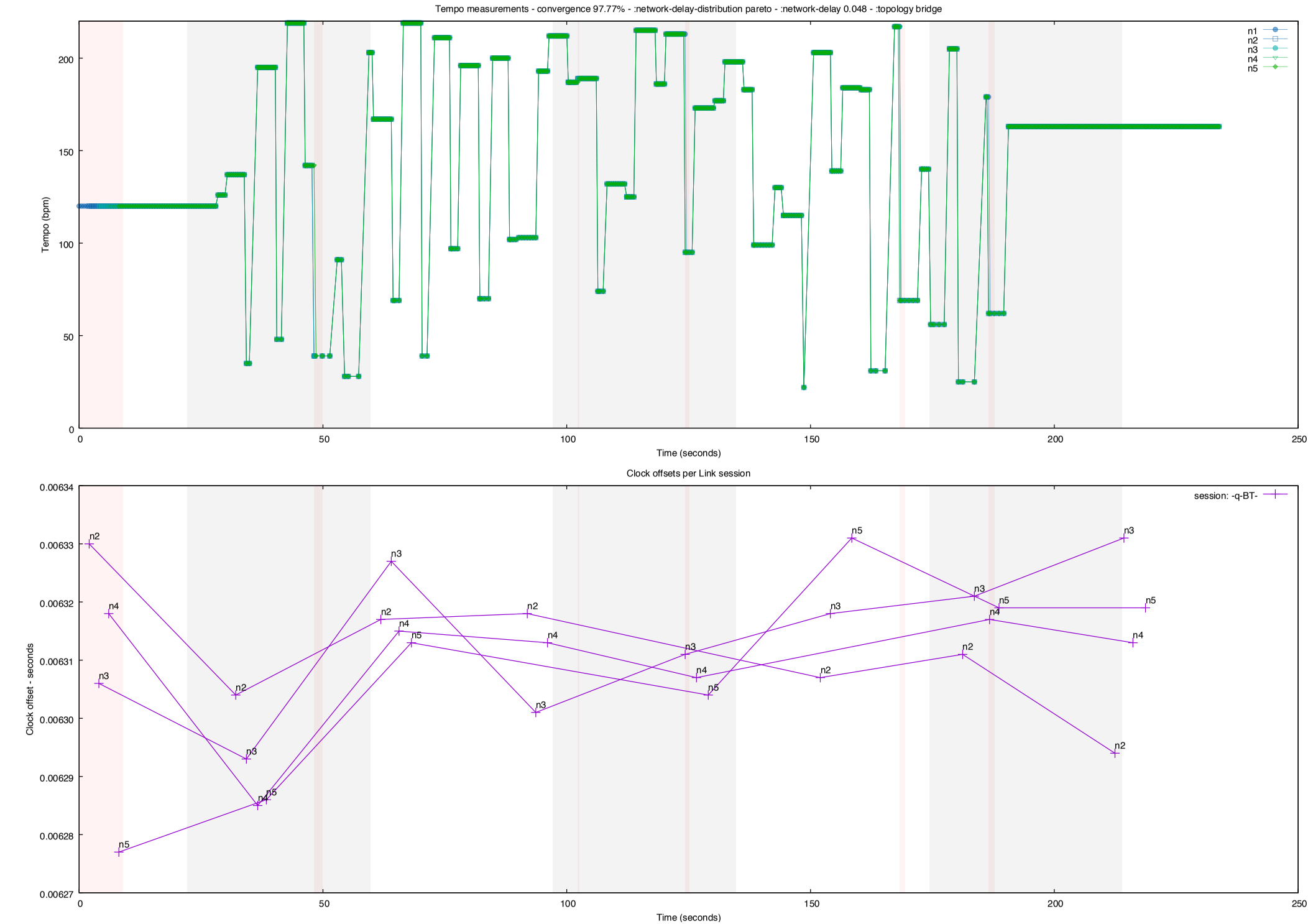

Tempo measurements - convergence 90.385% - :network-delay-distribution constant - :network-delay 0.048 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: ei3-d-T

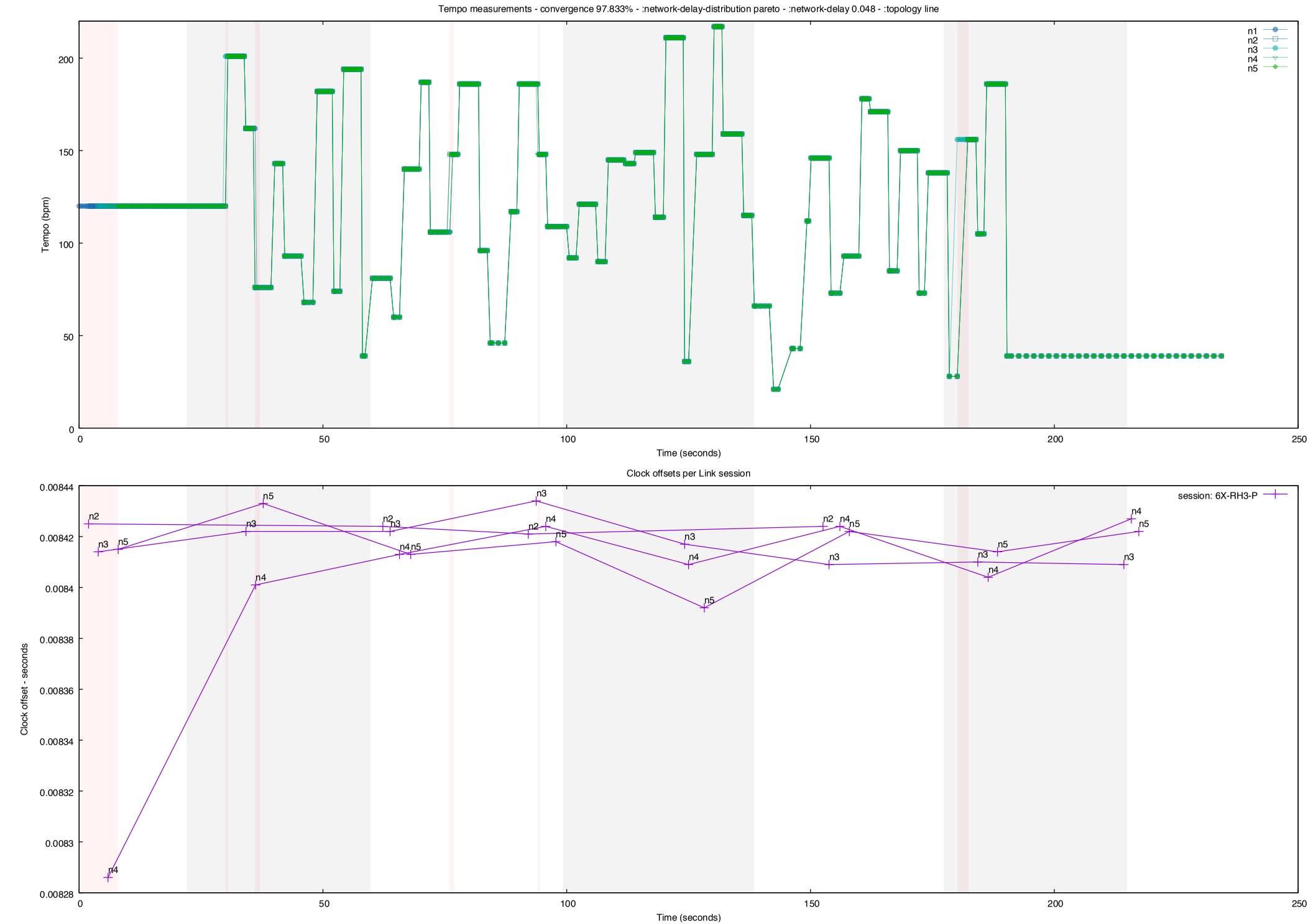

Tempo measurements - convergence 89.17% - :network-delay-distribution normal - :network-delay 0.048 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: v-4

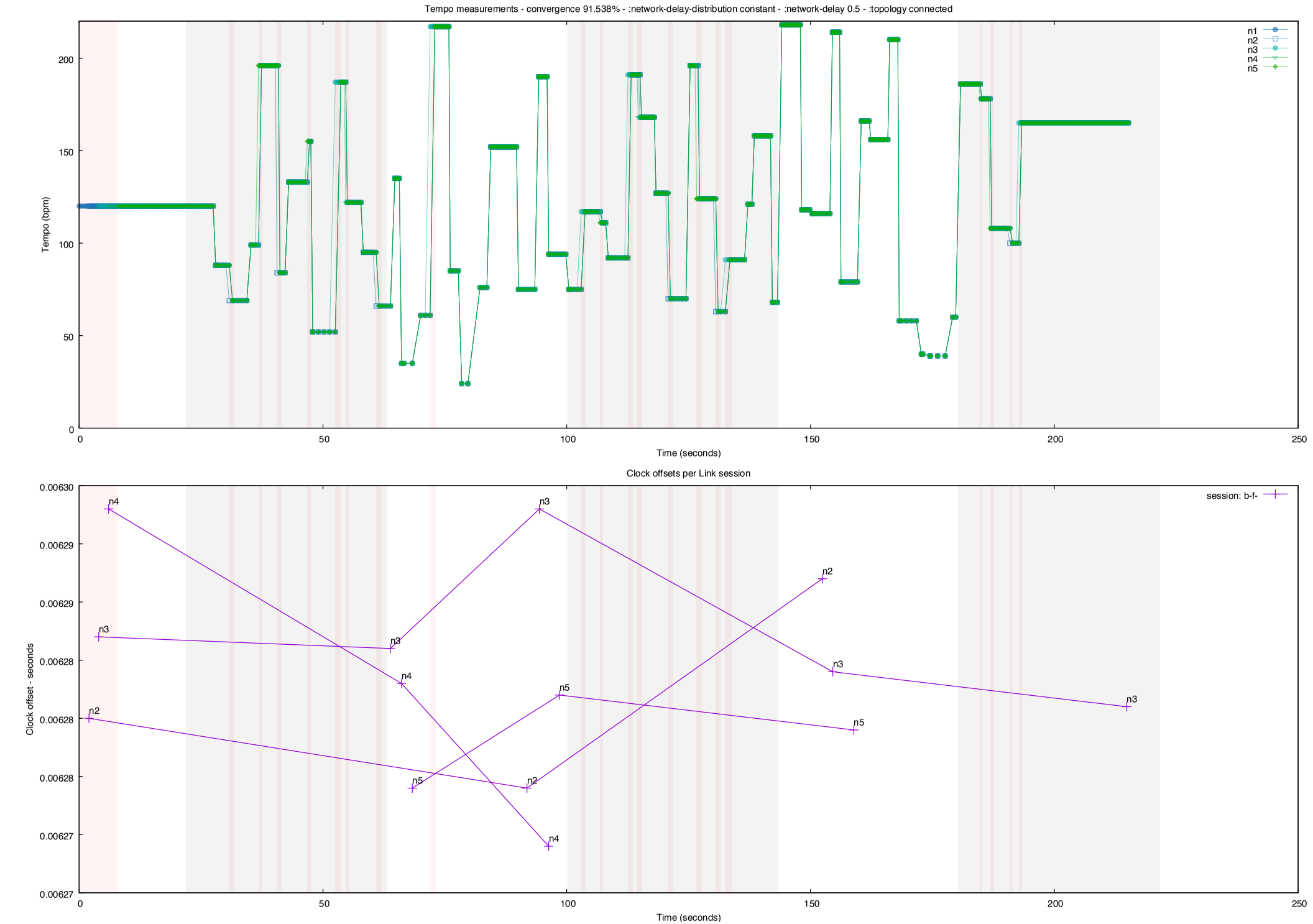

Tempo measurements - convergence 93.432% - :network-delay-distribution normal - :network-delay 0.048 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -EZJjRg-

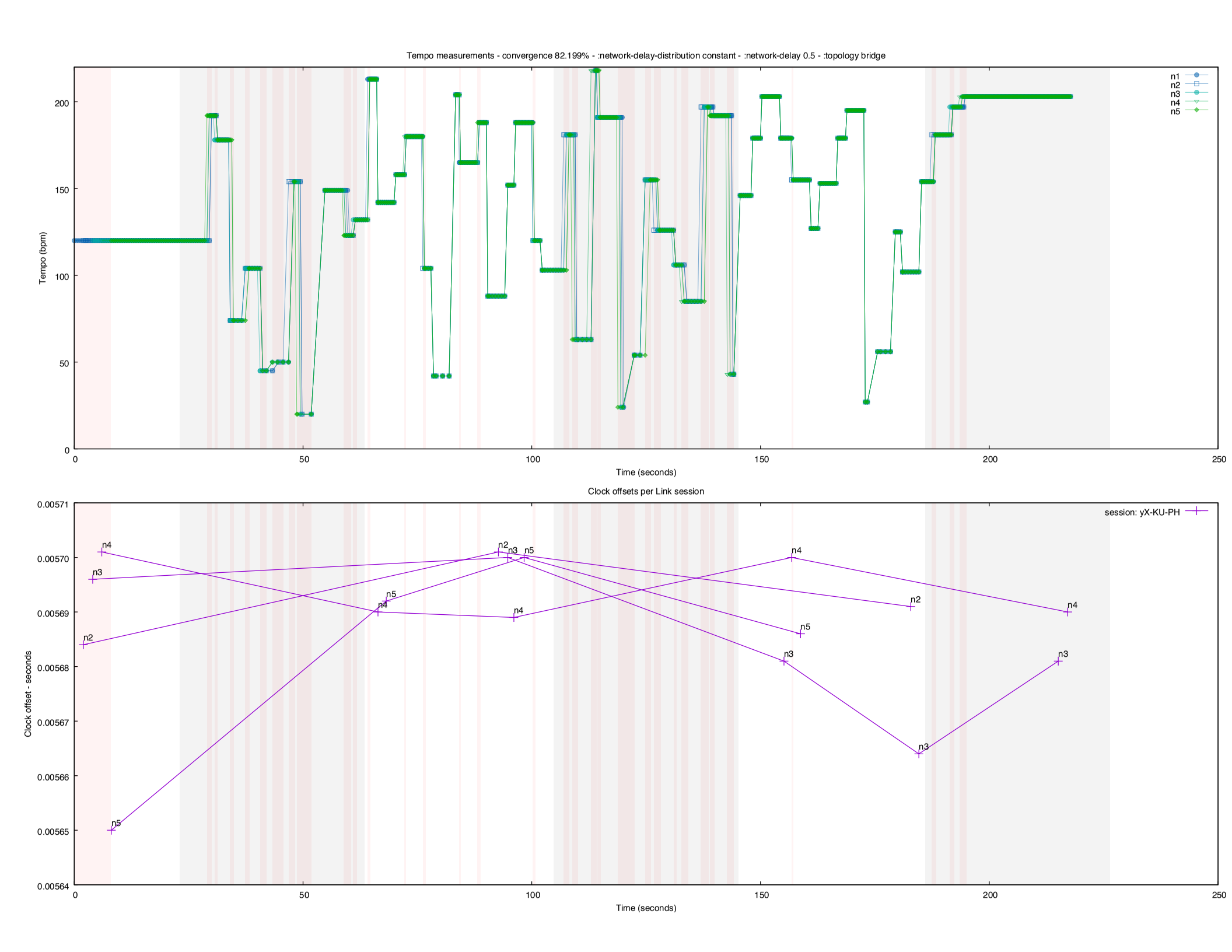

Tempo measurements - convergence 95.088% - :network-delay-distribution normal - :network-delay 0.048 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
session: W-x6NsD-
n2n3n4n5
n3
n4
n5

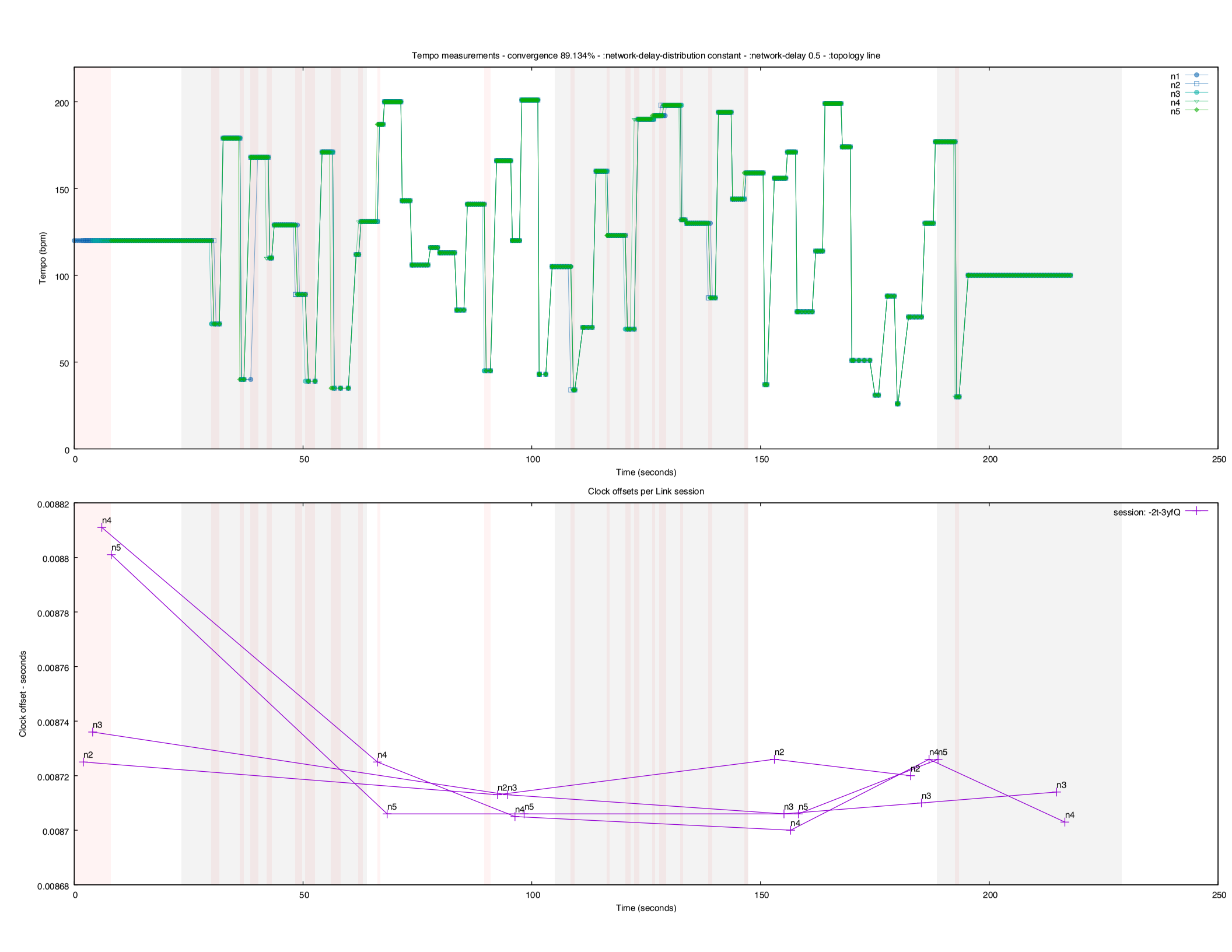

Tempo measurements - convergence 100.0% - :network-delay-distribution pareto - :network-delay 0.048 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -x-U-ko

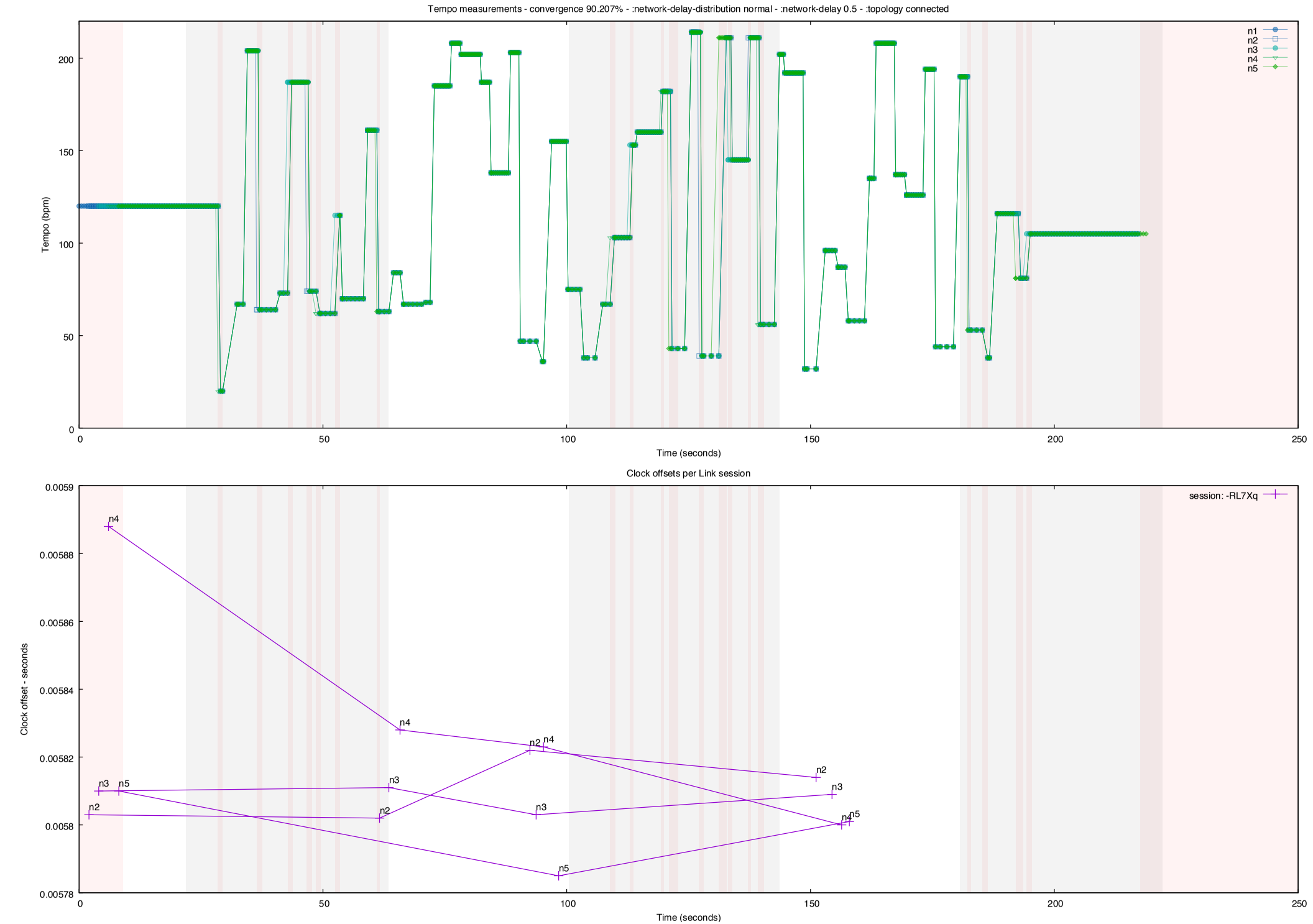

Tempo measurements - convergence 97.77% - :network-delay-distribution pareto - :network-delay 0.048 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -q-BT-

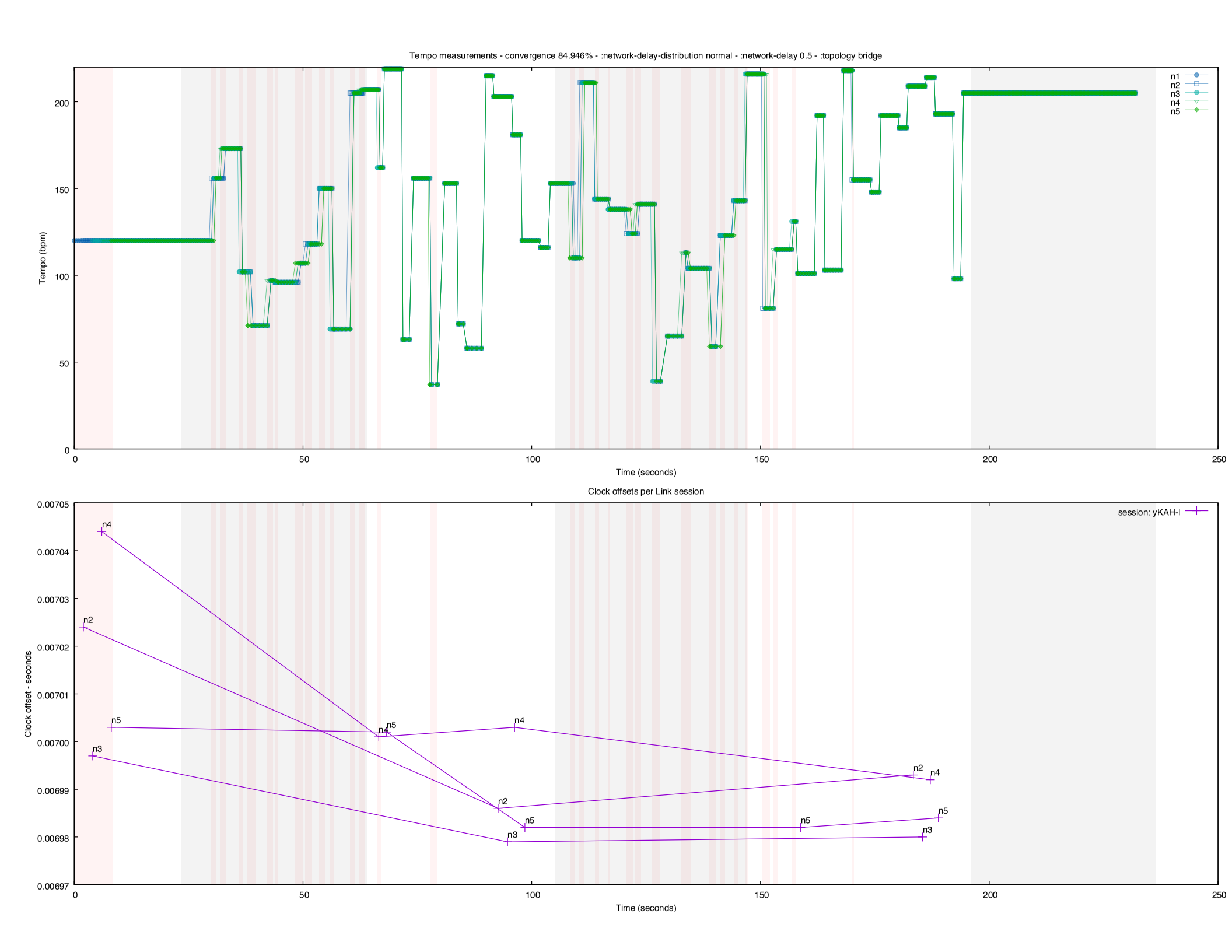

Tempo measurements - convergence 97.833% - :network-delay-distribution pareto - :network-delay 0.048 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: 6X-RH3-P

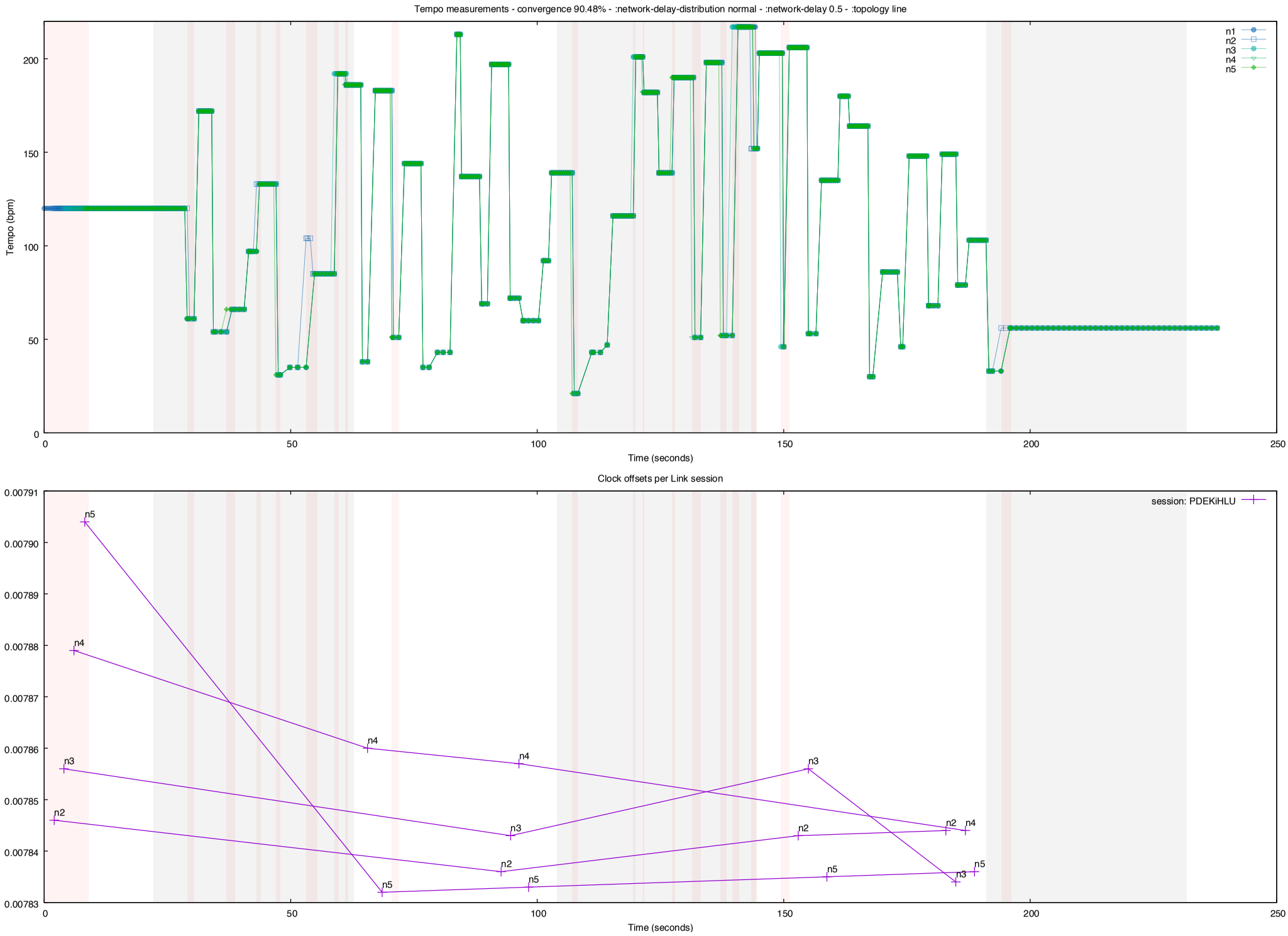

Tempo measurements - convergence 91.538% - :network-delay-distribution constant - :network-delay 0.5 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: b-f-

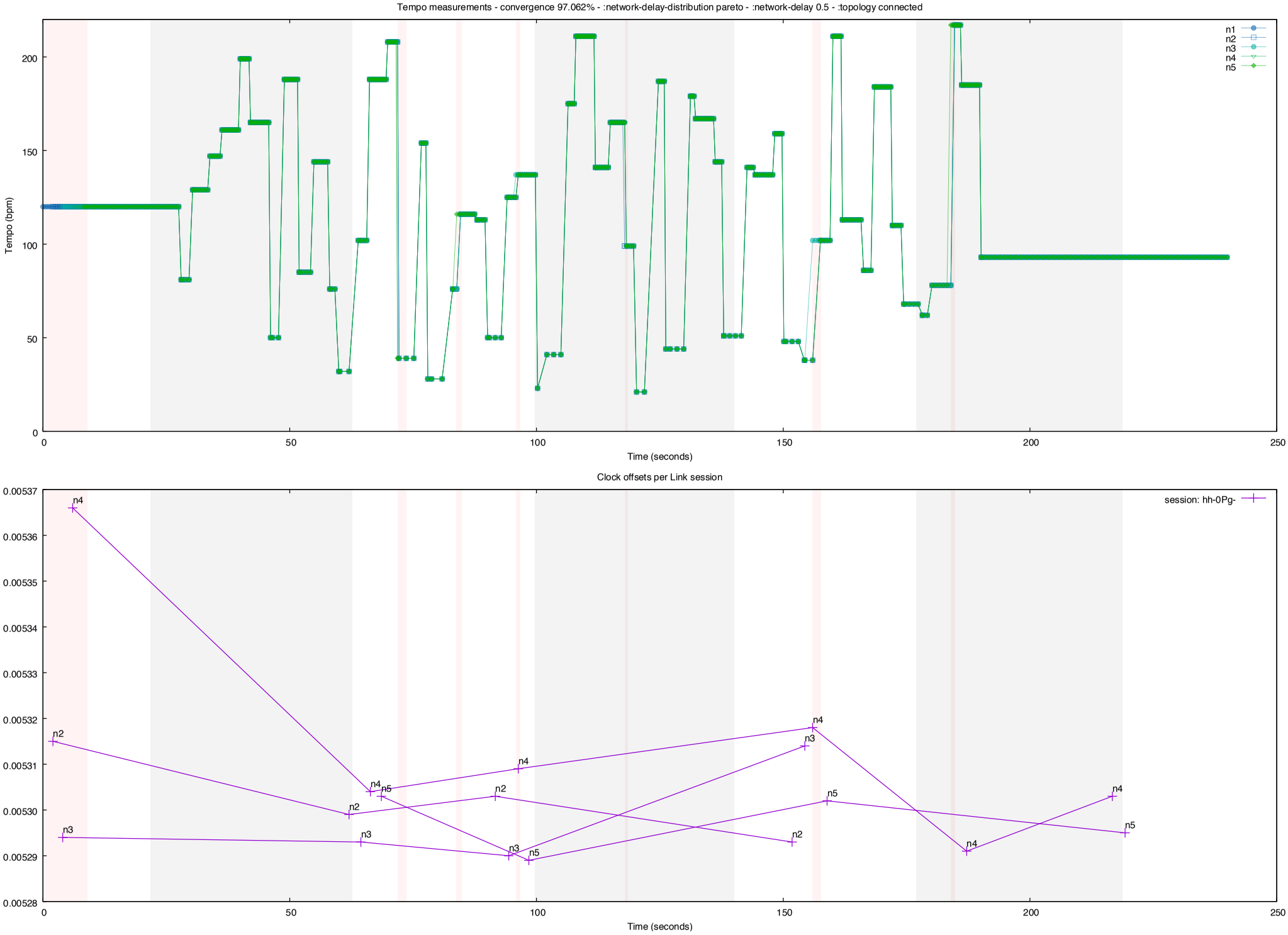

Tempo measurements - convergence 82.199% - :network-delay-distribution constant - :network-delay 0.5 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: yX-KU-PH

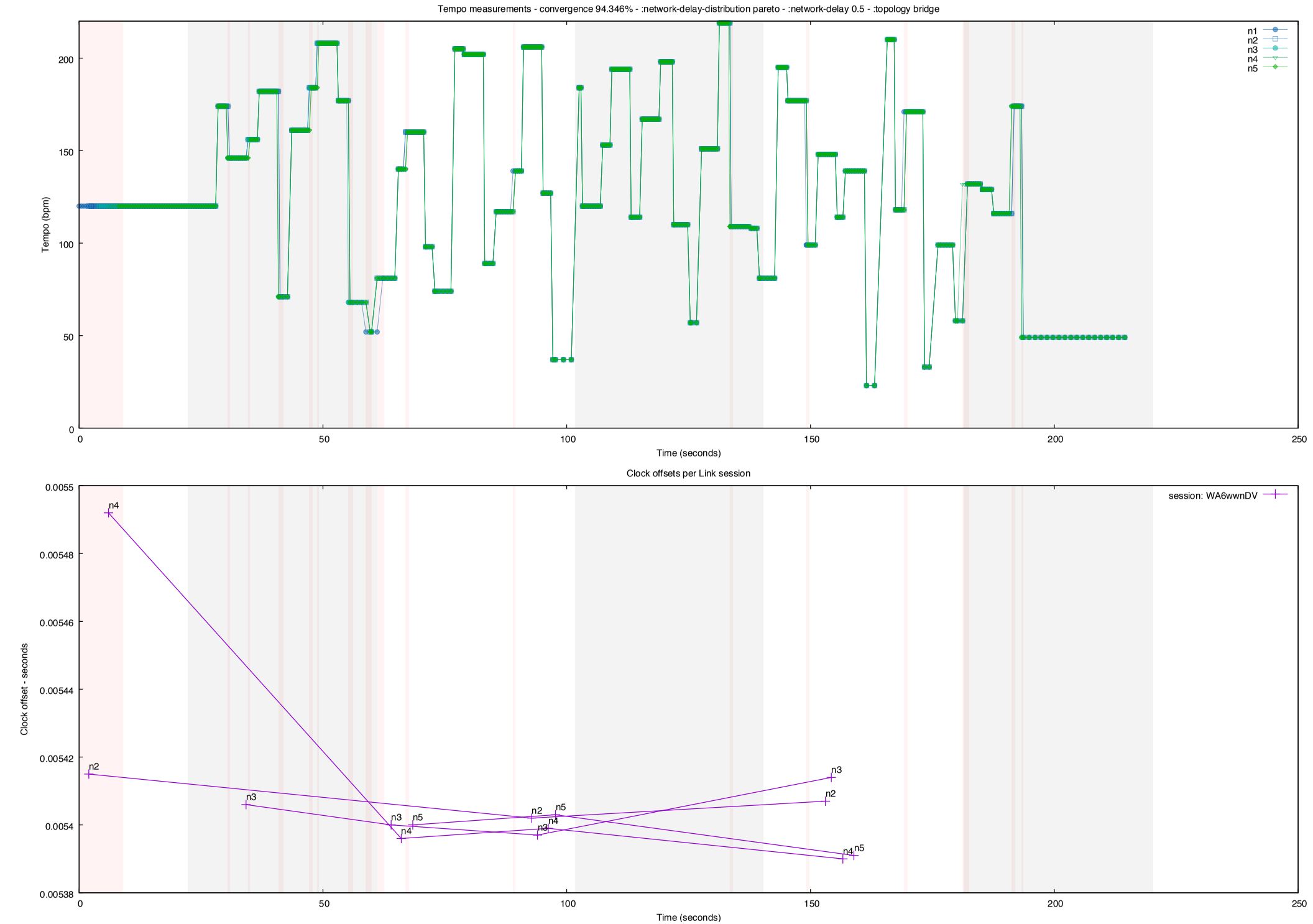

Tempo measurements - convergence 89.134% - :network-delay-distribution constant - :network-delay 0.5 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: -2t-3yfQ

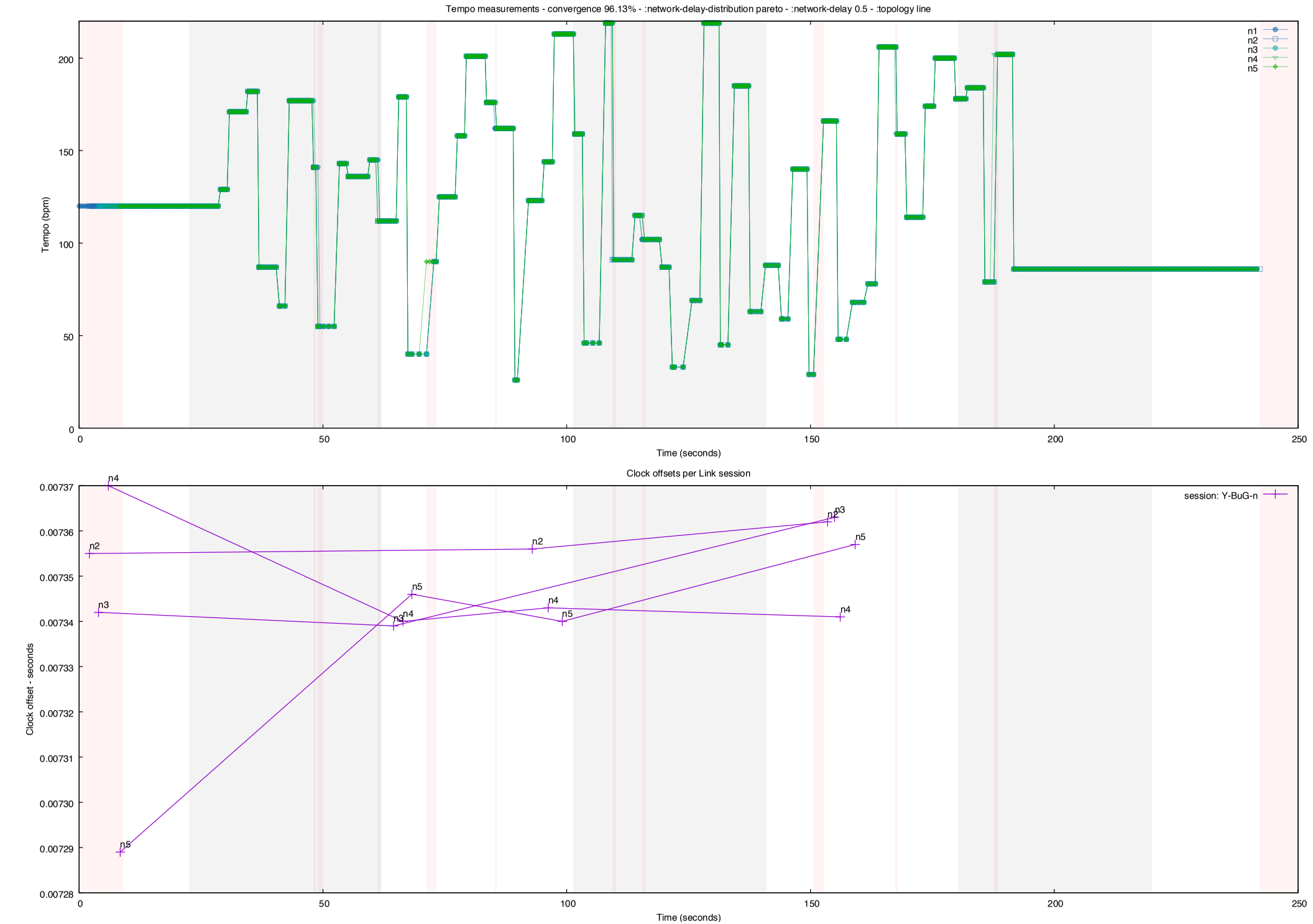

Tempo measurements - convergence 90.207% - :network-delay-distribution normal - :network-delay 0.5 - :topology connected
n1
n2
n3
n4
n5
Tempo (bpm)
Time (seconds)
Clock offsets per Link session
session: -RL7Xq
Clock offset - seconds
Time (seconds)


Tempo measurements - convergence 84.946% - :network-delay-distribution normal - :network-delay 0.5 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
session: yKAH-I


Tempo measurements - convergence 90.48% - :network-delay-distribution normal - :network-delay 0.5 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: PDEKiHLU


Tempo measurements - convergence 97.062% - :network-delay-distribution pareto - :network-delay 0.5 - :topology connected
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: hh-0Pg-


Tempo measurements - convergence 94.346% - :network-delay-distribution pareto - :network-delay 0.5 - :topology bridge
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: WA6wwnDV


Tempo measurements - convergence 96.13% - :network-delay-distribution pareto - :network-delay 0.5 - :topology line
Tempo (bpm)
Time (seconds)
n1
n2
n3
n4
n5
Clock offsets per Link session
Clock offset - seconds
Time (seconds)
session: Y-BuG-n

5 Analysis

5.1 Musical Consistency

The previous results are summarized in the following table:

Musical consistency			
delay	connected	bridge	line
0ms	98.644%	96.212%	96.942%
48ms constant	89.858%	91.571%	90.385%
48ms normal	89.17%	93.432%	95.088%
48ms pareto	100.0%	97.77%	97.833%
500ms constant	91.538%	82.199%	89.134%
500ms normal	90.207%	84.946%	90.48%
500ms pareto	97.062%	94.346%	96.13%

These show that the Link protocol is generally resilient to different network conditions. No test run observed any major divergence and musical consistency was generally good under the zero delay and pareto conditions.

There was considerable difficulty in obtaining precise measurements given the lack of configuration very little in the way of callbacks, especially regarding clock synchronization. This may account for some of the variation, however the graphs reflect that for the most part, 5 nodes were maintaining synchrony.

5.2 Effects of delay distributions on clock synchronization

In order to simulate network delays of various types the Jepsen framework uses the NetEm (network emulation) module as part of the Traffic Control package on Linux[10]. This allows for outgoing packets from a node to be delayed by a given amount, according to different statistical distributions.

- constant - each packet is delayed by a fixed amount
- normal - packets are delayed by a Gaussian distribution centred on the given time
- pareto - given time parameter defines the duration over which a long tail distribution is calculated.

Of these, the pareto distribution is the most similar to the self-similar "heavy tail" delays found in real-world WiFi networks[3][23], however the

constant and normal distributions are useful indicators for other types of network conditions. For example, in [21] delays to secondary users caused by interference (contention) are linear with respect to their distance from the access point. More constant delays are therefore useful to indicate the performance of Ableton Link for the lower bounds of latency under these kinds of conditions.

The Ableton Link library code contains an undocumented upper limit on clock synchronization round trip times: Any measurement which takes longer than 50ms will be rejected and retried. If more than 5 measurement packets (again, an undocumented limit) are "in flight" during the run of measurements, the run will be cancelled for that round. The process will then wait for the next 30 second interval to pass before attempting to synchronize again.

This means that any sustained period of delays above 50ms will result in synchronization measurements not being taken. This can be seen in the results above, for the 500ms delay condition - no synchronization events occur in the grey shaded areas which represent the network conditions being applied. Exceptions to this may occur when the testing framework is not able to execute the traffic shaping command on a node at the start of a nemesis period.

5.3 Symmetric and asymmetric delays

For delays below the 50ms RTT threshold, clock synchronization events will still take place, however the clock synchronization algorithm does not reflect the presence of delays in cases where both the forward path (leader) and the reverse path (peer) have delays to their outbound packets. This is because a symmetric delay will centre around a measurement of zero (or the original offset as if no delay were present).

Aside from being a source of confusion during early investigation, this is actually a desirable condition in Ableton Link; The point at which peers are judged to be synchronized is their "time at speaker" and given that the local computation related to timestamping events does not require network calls then any offset introduced by network latency should (ideally) not be taken into account.

Introducing an asymmetric delay in the tests on all peers except the leader causes synchronization events to be offset by half of the given delay time. In the examples above this is typically around 25ms negative offset for the 50ms one-way delay. Placing the asymmetric delay on the leader only causes the offset to "flip" to a 25ms positive offset. In the results this

can be seen as a movement from the typically offset of between 5-7ms (it is suggested that this is a result of processing overheads in the containers) to an offset of around -20ms.

5.4 Bandwidth usage and message complexity

Using the logging features of the iptables program, the number of packets sent between nodes during a test run (connected network, no delay) are shown as follows:

Packet count / Bandwidth					
from/to	n1 (leader)	n2	n3	n4	n5
n1	"-/-"	1829/236K	1823/235K	1822/235K	1826/236K
n2	1826/225K	"-/-"	1502/203K	1498/202K	1504/203K
n3	1818/224K	1500/203K	"-/-"	1492/201K	1498/202K
n4	1814/224K	1493/202K	1489/201K	"-/-"	1491/201K
n5	1816/224K	1497/202K	1493/202K	1488/201K	"-/-"

Here we see that the leader (n1) handles roughly 17.5% more than the other peers in the connected case with no delays applied. This is because the Ableton Link library will prefer a connection to the leader during a clock synchronization event. If the leader is not accessible from the peer, the next neighbouring peer ordered by ID is chosen.

This does raise a theoretical concern with regards to scalability as the leader handles proportionally more traffic, however in practice the group is unlikely to reach a size required to cause stability issues in a local network.

Comments in the codebase indicate that future work to choose a synchronization peer based on the local topology are being considered.

5.5 Resilience during periods of packet loss

The design of Ableton Link opts for UDP messaging throughout. Somewhat surprisingly for a system without a reliable messaging layer like TCP, the Link protocol maintains musical consistency under conditions of heavy packet loss. This may be in part due to the high level of redundancy in broadcasting updates - each node will re-broadcast the relevant state changes after receiving a "newer" event containing a change to the local state. In addition, each node will broadcast state at 5 second intervals. At the expense of some bandwidth, this approach does appear to have merit when looking at the results of the tests.

5.6 Comments on mutable timelines and temporary divergence

A number of applications that rely on timing do so using the system clock. While this is relegated to being an implementation detail in the literature, there is a growing agreement in the software engineering community that using the system clock is an unreliable source for measuring elapsed time, in part due to it's mutability. This has been the source of a number of major software outages and failures[9].

A proposed method which offers a reliable alternative for measuring elapsed time is the use of monotonic clocks. This are based on a monotonically increasing counter and have the desirable property that they will never be adjusted backwards or run at an artificially fast or slow rate, unlike system clocks under the control of NTP or similar synchronization methods.

Monotonic clocks are used by Ableton Link[7] to avoid such issues. This is important because the ordering of events and the election of a leader in a session are determined using timestamps. Using a mutable source of timestamps such as the system clock would affect the ability of the system to maintain correct ordering.

In addition to peers maintaining their own monotonic clock timestamp, the clock synchronization process determines additional offsets (deltas) relative to the leader of the session. Unlike the monotonic counter however, these offsets are necessarily mutable and can have the effect of moving a node "back in time" relative to another peer.

Algorithm 3 Temporary Divergence

1:

```
# Ping/pong measurement has hardcoded cap of 50ms for RTT
# Assuming RTT / 2 for latency calculation, this allows
# for a deviation of +/-25ms either side of leader

# Node A - @nodeA.delta == 25ms
@nodeA.set_tempo(120)
@nodeA.bcast({:tempo => 120, :global_host_time=> (99.975 + 0.025)})
# clock synchronization event - @nodeA.delta == 0m
@nodeA.set_tempo(140)
@nodeA.bcast({:tempo => 140, :global_host_time=> (99.975 + 0.0)})
# 140 is set locally (timestamp not checked for local updates)
```

```
# Node B
@nodeB.receive({:tempo => 120, :global_host_time => 100.0}) do
  if received_global_host_time > current_tempo.global_host_time
    @nodeB.set_tempo(received_tempo)
    @nodeB.bcast({:tempo => received_tempo,
                  :global_host_time => received_global_host_time})
  end
end

# 120 has a higher global host time
# 140 is therefore rejected, introducing temporary inconsistency

@nodeA.receive({:tempo => 120, :global_host_time => 100.0})
# this is newer that the timestamp for @nodeA's current tempo,
# therefore @nodeA will adopt it

# If @nodeA.receive(120) is dropped or lost when received by
# @nodeC, then @nodeC will adopt 140 as the tempo, however it
# should still converge.
```

Normally conditions such as these might pose a major issue to the consistency of the protocol, however eventually consistency is usually restored as state is broadcast at regular 5 second intervals (effectively rounds) and on subsequent state updates. This means that the agreement on tempo among the session is unlikely to diverge for more than 5 seconds.

Even so, this inter-round divergence is an area that would need attention if this protocol were to be adapted to environments with more strict requirements e.g. factory/warehouse automation.

At the expense of an increase in message size, a potential solution to this issue is offered by Hybrid Logical Clocks[12] proposed by Kulkarni et al.

Algorithm 4 Tempo changes with Hybrid Logical Clocks

1:

```
# Node A - @nodeA.delta == 25ms, @nodeA.logical_time = 0
@nodeA.set_tempo(120) # @nodeA.lt += 1
@nodeA.bcast({:tempo => 120,
              :logical_time => 1,
              :counter => 0,
```

```
              :global_host_time=> (99.975 + 0.025)})
# clock synchronization event - @nodeA.delta == 0m
@nodeA.set_tempo(140) # @nodeA.lt += 1
@nodeA.bcast({:tempo => 140,
              :logical_time => 2,
              :counter => 0,
              :global_host_time=> (99.975 + 0.0)})

# Node B - @nodeB.logical_time = 0
#        - @nodeB.global_host_time = 100.1
@nodeB.receive({:tempo => 120,
                :logical_time => 1,
                :counter => 1,
                :global_host_time => 100.0}) do
  @nodeB.logical_time = max([self.logical_time,
                             received_logical_time,
                             self.global_host_time])

  if self.logical_time.unchanged? && \
     self.logical_time == received_logical_time
    self.counter = max(self.counter, received_counter) + 1
  elseif self.logical_time.unchanged?
    self.counter = self.counter + 1
  elseif self.logical_time == received_logical_time
    self.counter = received_counter + 1
  else
    self.counter = 0
  end

  @nodeB.set_tempo(received_tempo)
  @nodeB.bcast({:tempo => received_tempo,
                :logical_time => 100.1,
                :counter => 0,
                :global_host_time => received_global_host_time})
end

# on receipt of tempo 140 at time 100.2
# ...
@nodeB.bcast({:tempo => received_tempo,
              :logical_time => 100.2,
```

```
                :counter => 0,
                :global_host_time => received_global_host_time})
# ...
```

Here the timestamp on the receiving peer allows for a causal ordering meaning that the later message is adopted.

6 Further work

The testing of distributed systems techniques applied to musical synchronization would primarily benefit from more systems to test against! Using the measures and techniques offered in this work it is hoped that an objective comparison with future libraries may be made.

Returning focus to Ableton Link, the testing approach may form the basis of some usable integration testing. In particular the use of containers (and Jepsen) to provide a reproducible testing environment is being adopted as part of a development workflow by several distributed systems such as Cockroach DB[18]. Combining this testing approach with the method of recording audio from each peer would allow for more objective measurements to be made with regard to clock synchronization.

Given the opportunity, it would be interesting to examine alternative techniques for clock synchronization that better accommodate asymmetric delays and also that take account of the network topology. One example would be the use of "coded probes" suggested by Geng et al[6] - by sending timing packets from server i to j in pairs with a known delay s between them, on being received at server j and the interval can reflect the presence of delays in transmission. As delays are often random in nature, the intervals closest to s can be used to increase accuracy.

Another area of interest would be to examine this work in the context of related fields, one in particular being factory automation, which requires synchronization between independent moving objects. Some of the approaches used in the Link library may be of use in such systems however there may be additional work required around the level and types of consistency that the library offers.

7 Conclusions

From the analysis performed in this work and anecdotally working with Link on other software projects I have found it to be an elegant, well designed

and well implemented solution to an interesting set of problems. Studying clock synchronization in the context of musical applications offers a practical introduction to the wider field, without the additional requirements in expertise that often come with analysing in the context of data storage systems.

Link has been shown to maintain a reasonable degree of consistency under adverse network conditions. This work has noted the impact of asymmetric delays under various distributions on the clock synchronization algorithms in use by Link. Bandwidth usage has been analysed and found to be higher on the leader in connected topologies, although data transfer in absolute terms is relatively low.

It is also proposed that the presence of a non-monotonic clock offset may introduce inconsistencies within rounds, however the regular retransmission of current state means that the session will converge eventually provided there are no permanent network partitions.

References

[1] Aaron, Sam. Sonic pi - the live coding music synth for everyone, 2013. [Online; accessed 18-August-2018].

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

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

[4] Chris Chafe and Michael Gurevich. Network time delay and ensemble accuracy: Effects of latency, asymmetry. In *Audio Engineering Society Convention 117*, Oct 2004.

[5] Cosgrove, Neil. Lnx studio, 2016. [Online; accessed 18-August-2018].

[6] 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.

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

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

[9] Habets, Thomas. gettimeofday() should never be used to measure time, 2010. [Online; accessed 17-August-2018].

[10] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, pages 18–23, 2005.

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

[12] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

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

[14] 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.

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

[16] 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.

[17] Reid Kei Oda. *Tools and Techniques for Rhythmic Synchronization in Networked Musical Performance*. PhD thesis, Princeton University, 2017.

[18] Poss, Raphael. Diy jepsen testing cockroachdb, 2016. [Online; accessed 17-August-2018].

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

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

[21] Pu Wang and Ian F Akyildiz. Improving network connectivity in the presence of heavy-tailed interference. *IEEE Transactions on Wireless Communications*, 13(10):5427–5439, 2014.

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

[23] Ying Jun Zhang, Soung Chang Liew, and Da Rui Chen. Delay analysis for wireless local area networks with multipacket reception under finite load. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–6. IEEE, 2008.

A Self assessment

The focus of this work was distilled over time, however the original impetus was related to my work on the open source music software Sonic Pi[1]. From an initial desire to add collaboration features to the software I began to explore the problem space. Private correspondence with Sam Aaron was useful in identifying the separate issues of co-located and geo-distributed performance.

With a view to implementing a custom solution to the co-located problem specifically, I began some preliminary designs. During this time I was introduced to the Ableton Link library which articulated many of the ideas I had already been considering, albeit with greater clarity and focus. As I began to investigate the Link library I became convinced that it was a well thought out solution to the co-located problem.

This led to preliminary work with the Link C++ library and as part of this project I began work on integrating this with the Ruby programming language (see code listings). Ruby was chosen as this is the primary development language for Sonic Pi and the creation of a wrapper should be of use in future integration work.

Many more hours were then spent working out how to instrument the Link codebase to allow for testing. With very few configuration options, the only path became to enable and parse the debug logs for information regarding clock synchronization events. These were largely undocumented

meaning that considerable time was spent on this stage in understanding the output.

With the C++ wrapper written, work moved on to setting up the tests in Jepsen. This was aided by generally excellent documentation on the Jepsen project, however the existing examples assumed the use of database client libraries which meant that working out how to handle more simple connection methods was a little more opaque.

Jepsen is written in Clojure which meant working in three quite different languages to produce this work (the others being C++ and Ruby). This was the source of some cognitive overhead which on reflection was not ideal, but given the choice again I would still work with the same libraries to do this kind of testing. I enjoyed working with Clojure and Jepsen in particular and I hope to work with these to devise more tests in future.

It is with some regret that only a single library was under test in this work, however after examining various solutions in the same space none of these were suitable for various reasons. These included reliance on a single operating system (OS X in the case of Landini[15]), or the need for graphical interfaces in the case of LNX Studio[5].

The final, and most significant, stage was spent in producing readable output from which to derive results. Inspiration was taken from the existing GNU Plot figures that are included with Jepsen to chart network performance, however all the code to produce the examples in figures here had to be written from scratch.

When dealing with parsing any text format, issues occur around the quality of the data. As a result the code contains a lot of incidental complexity which reflects the fact that the debug logs weren't written with instrumentation in mind. This has given me some insight into how one might choose to instrument a similar codebase in future, to aid accurate testing.

Finally while initial drafts of the work focused on variations in topology and delay, following advice from my supervisor I took the time to investigate different distributions of delays. This also highlighted to me the surprising (to me, at least) behaviour of the clock synchronization under symmetric delays. This meant that further work on the tests was required to ensure that a consistent leader was chosen for each test run and that delays were only applied to outbound packets on other nodes.

While the results presented here are not groundbreaking, I do hope that by offering a practical set of criteria and test bed that it will aid further research in this area. Thorough analysis of any distributed system is a difficult task and I have learnt a great deal about this process as a result.

B Running the tests

```
# clone project and note the path
git clone https://github.com/xavriley/jepsen-ableton-link.git

# clone the Jepsen project to another folder
git clone https://github.com/jepsen-io/jepsen.git#0.1.9
cd jepsen/docker
export JEPSEN_ROOT="/path/to/jepsen.link" # as noted above
./up.sh --dev

# in another window
docker exec -it jepsen-control bash
# once the container has booted
lein run test --time-limit 180 --no-teardown \
--topology line --network-delay 0.5 \
--nemesis-duration 30 --network-delay-distribution constant
```

Once that has completed, run the following in the root of the jepsen.link project:

```
ruby tempo-grapher.rb
```

This will generate the graphs in

```
./figures_for_publication/_name_of_autogenerated_test_output_directory_/plot.pdf
```

following each test run.

C Code listings

The following sections outline the code written in support of this work.

C.1 `ruby_ableton_link` gem

The code is available at `https://github.com/xavriley/ruby_ableton_link`.

The majority of interest is contained in the `ext/ableton_link/ableton_link.cpp` file. This contains the wrapper code for the Ableton Link C++ library.

The wrapper is implemented using the Rice project which provides convenience functions for interoperability between C++ and Ruby codebases: `https://github.com/jasonroelofs/rice`

The file used for the test procedure is `bin/server` which contains a simple TCP server to receive messages containing tempo information while printing the status of the session on each beat.

C.2 `jepsen.link`

The code is available at `https://github.com/xavriley/jepsen-ableton-link`.

This is a Clojure project, using the Jepsen framework. All code relating to the testing is contained in the `src/jepsen/link.clj` file.

To produce the graphs the `tempo-grapher.rb` file is used. This code is all related to the parsing of log files and the production of a `gnuplot` command. The data produced by the parsing is saved to files and copied to the output directory in case further analysis is required.

D Professional issues

When assessing software solutions for synchronizing musical time, it is admirable that Ableton Link attempts to do so using permissive licensing and a committment to cross platform support. In doing so they allow for the possibility that freely available software may be used for music production without onerous requirements on the user to learn a particular programming language or maintain a specific environment in order to be productive. I would hope that this has a democratizing effect on the landscape for music production, particularly with regard to some of the academic software output in this field. These shared projects, while excellent in many respects, often have a high barrier to entry that musicians with a lack of appropriate resources would struggle to overcome.

Without more effort on the part of publically funded institutions to target a non-academic userbase, the danger is for a tendency toward an implicit elitism in their outputs. Software and music is then written for, as received by, audiences from within the institutions without regard to how the public life of the wider communities can be enriched.

With a void of accessible, open source (OSS) alternatives the space is left for private companies and individuals to "productize" the work done in academia, with high costs for the end user. Music forms a hobby (in many cases a passion) among a large percentage of the population - indeed a *dislike* of music is so rare that it is deemed a medical condition - textitmusical anhedonia - affecting around 3-5population exist a large group who invest a great deal of time and energy into music production. Given the dominance

of the private market in digital music software, this often entails a financial investment too and a dependence on expensive tools.

Instead of using a passion for digital music making in a constructive way, proprietary software has no incentive to educate the end user beyond making them comfortable with their proprietary interface. Music has offered a gateway into computer science for many current practioners (annecdotally, the creator of Clojure Rich Hickey started as a music graduate programming applications for the Atari ST) - if the programming opportunities are not provided then it's possible that a generation of potential programmers will miss out on these opportunities.

To this end, open source projects like Sonic Pi[1] are aiming to bridge this gap by targetting school children alongside professional users. It does so by packaging the work of other OSS projects such as SuperCollider, Qt, QScintilla and the Ruby programming language into a cross-platform environment with a wide reach and a large userbase. My own contributions to that project have been with the aim of making new tools of expression more freely available to those with an interest in creating.

This is where the model of Ableton Link, despite being written by a proprietary software company, is forward looking in the creation of an open standard to solve the issue of synchronization.

E Acknowledgements

I would like to record my thanks to a number of groups and individuals, without whom this work would not have been possible.

Firstly, to the open source community and in particular the Ruby and Clojure projects and practicioners I have interacted with. These opportunities have enriched my career immeasurably. In particular I'd like to thank Sam Aaron for his insightful comments, stimulating discussions and boundless energy in working on Sonic Pi, along with the other contributors.

It's also fitting that the Kyle Kingsbury's work on the Jepsen project first piqued my interest in distributed systems, so I'm pleased and proud to have been able to continue that in the presented work. Honourable mentions also go to Ableton for developing Link and doing such a good job of it, along with gnuplot and vimlatex for making the test results look half decent.

My thanks also to my work colleagues at Heroku for supporting me during a part-time pursuit of a Masters degree and to Salesforce for the generous funding policy.

To Gregory and the teaching staff at RHUL, I'm forever grateful that

I was given the opportunity to study with you and I've found it rewarding and challenging in the best possible way.

Finally, to my friend David Bamber for introducing me to the idea that programming was for me, to my family for unerring support and to my wife Em who has been consistent, persistent and fault tolerant throughout.