Response Letter

We would like to thank the reviewers for the thoughtful comments, questions, and criticism.

We accommodate all clarifications and answers in a final version of the paper. To help the reviewers find the modifications of interest, we provide a link with a *diff* between the versions, as follows:

`https://github.com/fsantanna-no/p2p-tml-paper/compare/690245...main`

Based on the overall comments, our focus on the final version was to improve the section on related work, and to make the limitations of our proposal more explicit.

For that matter, we expanded Section "2. Related Work" by splitting it in two subsections "2.1. Symmetric Distributed Applications" and "2.2. Software Time Machines". We also rewrote Section "3.2.4. Middleware Summary" to discuss its limitations.

Next, we address each individual comment from all reviewers as follows... (Some comments might be reordered.)

Reviewer A

Regarding the section on related work (comment A.1):

Comment A.1

> The related work section could be expanded to include more recent work on peer-to-peer systems and distributed consensus algorithms;

We now expanded the end of Section "2.1 Symmetric Distributed Applications" to put our work into the perspective of classical distributed consensus algorithms:

> "Finally, note that classical distributed consensus mechanisms focus on fault tolerance [8], but not on real-time symmetric behavior. These algorithms typically rely on leaders [9], which require an election period that does not cope with our symmetric real-time requirements. In this sense, these algorithms could be considered to decentralize servers in Croquet or GALS, but not to decentralize symmetric peers with high churn rates, as we propose in this work. In contrast, our proposal requires a time machine to reapply reordered operations, since we cannot guarantee strong consistency [10]."

We now include recent references on consensus mechanisms:

- [2019] "Concurrency: the works of Leslie Lamport"
  - Leslie Lamport, Robert Shostak, and Marshall Pease
- [2020] "Paxos vs Raft: Have we reached consensus on distributed consensus?"
  - Heidi Howard, Richard Mortier
- [2017] "Verifying strong eventual consistency in distributed systems"
  - Victor Gomes, ..., Alastair Beresford

We also included a new reference for Croquet that has been published recently:

- [2022] "An experiment in live collaborative programming on the croquet shared experience platform"

– Yoshiki Ohshima, . . . , David A. Smith
– The last author is the first author of the original Croquet paper from 2005.
– They reimplement the original Croquet in JavaScript to run in modern computers. In their Section 2, they describe the network architecture that we discuss in our Figure 2.
– They also refer to CRDTs and derivatives as the most recent alternative with respect to real-time collaborative protocols (they roughly cite the same papers we do).

Regarding the limitations (comments A.2-A.3):

Comment A.2

> <. . . > The paper could discuss the limitations of the proposed approach and potential future work.

We now discuss the limitations of our work as follows. . .

- 1. Introduction:

  "We target small peer-to-peer networks, in which nodes are only a few hops away from each other. This ensures that events can span the whole network in a reasonable time to preserve the real-time behavior of applications. We also assume that peers are non-malicious in the sense that they do not generate erroneous events."

- 3.2.1. Event Dissemination:

  "Considering the peer-to-peer network as a whole, each event packet is replicated to all neighbours of all peers. This results in a theoretical limit of quadratic messages if all peers were connected to all peers. In contrast, as discussed in Section 2.1, a centralized solution only requires a packet to be sent to a server, which then broadcasts it to all clients."

- 3.2.4. Middleware Summary:

We discuss the middleware limitations as follows: contiguous unique identifiers, deterministic API, non-malicious peers, and static memory footprint. The text is too long to fit here and can be found in the paper or link above.

Comment A.3

> <. . . > The paper assumes non-malicious peers. What security measures or modifications would be necessary to protect against malicious peers attempting to disrupt the synchronization or inject false events?

We assume non-malicious peers in the network. This is a fundamental limitation also shared by the alternatives discussed in related work. Even with CRDTs, nothing prevents a malicious peer to inject 1M events per minute to break the application.

We now point this limitation in the Introduction, along with a reposition to target small networks, implying more control over participants:

> "We target small peer-to-peer networks, in which nodes are a only few hops away from each other. This ensures that events can span the whole network in a reasonable time to preserve the real-time behavior of applications. We also assume that peers are non-malicious in the sense that they do not generate erroneous events."

In Section "3.2.4. Middleware Summary", we now also explain how this limitation emerges in our API:

> "As detailed in Section 3.2.1, note that the API to disseminate an event in the network is as simple as a variable assignment. Therefore, a malicious peer could SPAM or inject erroneous events to break the application. Again, this is a fundamental limitation that is also shared with the alternatives discussed in Section 2."

Regarding the middleware & protocol (comments A.4-A.6):

Comment A.4

> While the API description is generally clear, some parts could be further clarified, particularly regarding the handling of dynamic memory allocation <...>

In Section "3.2.4. Middleware Summary", we now also illustrate how to accommodate dynamic allocation over static snapshots:

> "Regarding the static limitation, we mentioned the possibility of a custom allocator, which is illustrated as follows:"

```
struct {
    // FIXED REGION
    int x,  y;   // position
    int dx, dy;  // direction speed
    // VARIABLE REGION
    char heap[HEAP_SIZE];
} G;

T* obj = my_malloc(G.heap, <size>);
<...>
my_free(obj);
```

> "The variable region is still technically static, but is dynamically reshaped internally with the custom allocator `my_alloc` & `my_free`."

Comment A.5

> <...> and the interaction between the application and the middleware during time travel. <...>

The application is frozen during time travel.

In Section "3.2.3. The Time Machine", we now include the paragraph as follows:

> "The step delay is dynamically adjusted such that the whole time travel loop takes at most 1 second to complete. Note that during this period, the application is frozen, and new inputs are enqueued for further processing."

Comment A.6

> <...> Suggestion: if possible please consider providing a more formal description of the time synchronization protocol, including any guarantees or assumptions about clock drift and network delays.

Even though we consider that this suggestion is relevant, we believe that a formal description of the protocol with its implications requires a dedicated work on its own.

Regarding the experimental evaluation (comments A.7-A.9):

Comment A.7

> <...> The experimental evaluation could be more extensive, including more diverse scenarios *(see A.8)* and a comparison with existing approaches

At the end of Section "4. Evaluation", we now compare the existing approaches discussed in Section "2.1. Symmetric Distributed Applications" with respect to the three aspects that we consider in our evaluation (i.e., recurrence of time travels, real-time pace of peers, correctness of unstable peers). The text is too long to fit here and can be found in the paper or link above.

Comment A.8

> <...> The evaluation uses a fixed network topology. How would the middleware's performance be affected in scenarios with highly dynamic topologies, where peers frequently join, leave, or experience intermittent connectivity? Have you considered any mechanisms to adapt to such changes?

Even though the network topology is fixed, it includes three typical variations of (a) straight bus lines, (b) rings, and (c) connected topologies. They all coexist in the same deployment with peers leaving and joining the network. In item "(c) Correctness of unstable peers", we make all 21 nodes to switch between online and offline for the same amount of time, including the peers in the middle of the network. This churn rate of 50% periodically creates full partitions in the peer-to-peer network, making the two opposites incommunicable for a considerable amount of time. Therefore, our unstructured design does provide the necessary mechanisms to adapt to such changes, since peers joins and leaves only impact direct neighbours. Our evaluation takes high churn rates into consideration.

Comment A.9

> <...> The evaluation focuses on simulations with up to 21 peers. How do you envision the middleware scaling to much larger networks, and what challenges might arise in real-world deployments with potentially hundreds or thousands of peers?

We now position the scope of our work to small peer-to-peer networks only:

> "We target small peer-to-peer networks, in which nodes are a only few hops away from each other. This ensures that events can span the whole network in a reasonable time to preserve the real-time behavior of applications."

As suggested in this paragraph, our design is not appropriate for too many hops, since each hop increases the event latency, which harms real-time behavior.

Reviewer J

Regarding the section on related work (comments J.1-J.5):

Comment J.1

> i) This can do with a subtitles to make it easier to follow the discussion. There appears to be two major parts in this section: (1) the discussion on the three selected works (Croquet, GALS, CRDTs) in comparison to this work and this Work and (2) related works on time machines. It would be beneficial to clearly delineate these two subsection, preferably with subtitles.

As suggested, we now split Section 2 in two:

- 2.1 Symmetric Distributed Applications: discussion on the selected works, and comparison with our proposal.
- 2.2 Software Time Machines: expanded and improved discussion on time machines.

Comment J.2

> ii) The introduction on the related works makes it appear like there are ideally 3 subsections: (a) how the network is organized, (b) how global time is determined, and (c) how events are propagated and applied. However, this three aspects are actually related to the discussion on the selected works compared to the proposed solution.

Now, to make clear that we have only two subsections, we briefly introduce 2.1 and 2.2 with a paragraph at the beginning of Section 2 (before 2.1 and 2.2):

> "This section first revisits existing solutions for symmetric distributed applications, namely Croquet [2], GALS [1], and CRDTs [3]. Then, we discuss software time machines that are built for either local or networked applications."

We now link the three aspects (a,b,c) with the selected works at the beginning of 2.1:

> "Even though existing solutions aim to provide symmetric behavior for distributed applications, they diverge in three key aspects as follows: a. How the network is organized. b. How global time is determined. c. How events are propagated and applied. Figure 1 compares three selected works regarding these aspects." <...>

Comment J.3

> –> There is a need to reorganize the related works section, with a proper introduction showing the two core aspects of the discussion, (1) and (2) as indicated in i), forming the introduction to the section.

Done, as detailed in J.1 and J.2.

Comment J.4

> iii) The discussion does not have a very smooth transition. The punchline of the related works section should ideally be the proposal of this work, which is brought out more clearly when discussed at the end of the related works to show how all the gaps identified will be solved.

To guide our discussion, we now explain how the text is organized at the beginning of Section 2.1:

> <. . . > "Figure 1 compares three selected works regarding these aspects, which we discuss and contrast with our approach next. At the end of this section, we also provide initial details on how our middleware works.

Comment J.5

> iv) The discussion on related work on time machines is rather abrupt. It does not have a proper introduction to the problem of time machines in relation to this work (or in general for that matter) but directly proceeds into an analysis of Fusion and Braid. –> What is aimed by this section? Answering this can help find a way of introducing the time machines aspect well to the reader.

The discussion was indeed abrupt and out of context, so we reshaped the new subsection 2.2. We included two new paragraphs at the very beginning, in which we discuss (a) what a software time machine is, (b) in which scenarios it appears, (c) what are the challenges to overcome, and (d) what are the common implementation techniques. The text is too long to fit here and can be found in the paper or link above.

Regarding the protocol, middleware, and peer-to-peer concepts (comments J.7-J.11):

Comment J.7

> i) As a proof-of-concept, it will be important to have p2p model/framework to give a clear picture to an expert reader and explain to a lay reader the place of this middleware in the overall p2p application model/framework. Has this been considered?

We now include a paragraph at the beginning of Section "3. The Middleware & Programming API" describing the P2P framework or, in other words, what we exactly mean by P2P:

> "The peers form a dynamic unstructured peer-to-peer network [3], and communicate mostly indirectly to each other. Events are flooded in the network graph via gossiping, i.e.: when a peer generates an event, it communicates to its neighbours, which communicate to their neighbours, and so on. Note that all peers execute the exactly same application and middleware, with no differences with respect to their roles and physical resources. Therefore, we can describe our peer-to-peer network as follows: (a) all peers have the same role and run the same software; (b) any peer can join or leave at any time; (c) events are only communicated with direct neighbours."

Comment J.8

> ii) This proposal uses broadcasting to synchronize peers. What is the implication of this assuming a large p2p network? Has this been considered?

The use of the term "broadcast" was inaccurate in the paper. Now, every occurrence of "broadcast" was substituted by "trigger" or "propagation" or "dissemination", depending on the context.

As was described in the Introduction,

> "Peers in the application form a dynamic network graph and communicate only with direct neighbours, as in typical unstructured peer-to-peer networks. Events are flooded in the graph and are triggered locally with a small delay to compensate the network latency."

What we meant is that the same effect of a broadcast is reached after the events are flooded in the network, but only through direct neighbour communication. Nevertheless, we no longer use this loosened notion of a broadcast.

We now also emphasize this distinction from centralized solutions (which do rely on broadcasts) in Section "2. Related Work":

> "Note how the central server is the only participant with knowledge about clients, which never communicate directly among themselves. This implies that the server must never fail, and also that periodic full broadcasts are the only possibility of spanning the whole network. In this work, we argue that in a peer-to-peer alternative any node can fail, and that broadcasts can be replaced by optimistic flooding." <...> "Note, that it is not necessary that all peers communicate directly with all other peers, but only indirectly via unstructured flooding." <...>

Section "3.2.1. Event Broadcasting" was renamed to "3.2.1. Event Dissemination" with the introduction rephrased as follows:

> <...>, "events *propagate* between peers with a timestamp scheduled to the future with an extra delta such that all peers are able to apply them in sync. To prevent *dissemination* cycles, each peer" <...>

To conclude, there are no full broadcasts in the protocol. Regardless of the network size, peers only communicate with direct neighbours. In our experiments, flooding takes 5 hops in the average, with a maximum of 14 hops (between nodes 0 and 20). As an example, with a *100ms* network latency, it takes *500ms* in the average to flood an event in the P2P graph. We detail these costs in Figure 6. Therefore, we do consider the implication of hops and latency in the protocol.

Comment J.9

> iii) How are the events, as well as the memory snapshots, stored for persistence in the p2p network, that is, is there a storage mechanisms used so that an offline who joins immediately picks up from where it had dropped off?

The full history of both events and memory snapshots are stored in all peers. There are no differences among peers with respect to their roles: any peer can fail and any peer can offer full recovery. Therefore, an offline peer that (re)joins the network already holds past snapshots, and can fully recover from any direct neighbour.

We added a remark at the beginning of Section "3. The Middleware & Programming API", when describing the middleware architecture with Figure 4:

> "Note that all peers execute the exactly same application and middleware, with no differences with respect to their roles and physical resources."

We now also explicit that snapshots are replicated, further in the same section:

> "As an optimization, the middleware takes periodic snapshots locally *at all peers* to avoid full simulation." <...> "In addition, a peer that rejoins the network recovers from a past snapshot it already holds, thus avoiding full simulation."

Comment J.10

> iv) The simulations were done on the same machine. This means that there is no actual considerations due to the effect of network challenges. Is this the case?

The evaluation requires a synchronized global clock that can measure differences in the order of a few milliseconds. This can be done reliably in a single machine, but not trivially across the network with the precision that we needed. We use Linux's NetEm to emulate network latency, but do not consider other network challenges. That being said, latency is the only network dependent parameter that we currently evaluate.

Comment J.11

> v) Although this was considering the middleware only, it would be important to talk about the impact in terms of network efficiency if any due to the middleware.

I'm not sure if I understand this comment correctly. In Section 4, we discuss the negligible CPU impact the middleware has over the application as a whole, which is 9us every 20ms. Regarding the network, the middleware only receives and forwards event packets among neighbours, in the same way typical flooding protocols behave.

We now include a discussion about packet forwarding at the end of Section "3.2.1. Event Dissemination":

> "Considering the peer-to-peer network as a whole, each event packet is replicated to all neighbours of all peers. This results in a theoretical limit of quadratic messages if all peers are connected to all peers. In contrast, as discussed in Section 2, a centralized solution only requires a packet to be sent to a server, which then broadcasts it to all clients."

Regarding grammar issues (comments J.12-J.13):

Comment J.12

> Figure 2 - GALS –> check the spelling for client

Fixed.

Comment J.13

<. . . > This should be . . . among all peers to be the “correct real time”,. . .

Fixed.