FTMesh: Efficient Fault Tolerance for Interactive Applications on Service Mesh

Undergraduate student, submission #2

1 Motivation

Interactive applications, e.g., video conferencing applications, online games, and augmented and virtual reality applications, impose near real-time bounds on processing latency. Violating these processing requirements can lead to user-visible quality degradation, and in the worst case can render application inoperative. Similar to other distributed applications, these services are increasingly built out of multiple microservices, all of which must work in concert to fulfill processing requirements. In this paper, we describe our ongoing research on developing *a generic fault-tolerance approach* that allows these applications to *meet their stringent processing deadlines in the presence of failures.*

Meeting processing latency requirements under these stringent processing bounds is challenging: failure recovery for applications such as video conferencing must take significantly less than a few-100 milliseconds, and the inability to do so can lead to user-visible artifacts. This recovery time is far below what current microservice orchestrators target, necessitating the development of new techniques.

Current approach: We investigated the failure recovery mechanism of Jitsi [3], an open-source video conferencing service. It is composed of multiple services, and frequently deployed using Kubernetes, thus meeting our goals. Figure 1(a) shows part of Jitsi's failure recovery mechanism: when a video bridge failure is detected, the health-checker notifies the conference controller, which selects an alternate video bridge (a target) to take over for the failed one, and synchronizes session state with the target. Once state synchronization is completed, the target bridge informs the controller, which then notifies the client, and they establish connections with the target. We observe from the timeline of this procedure in Figure 1(b), that re-establishing client connections (which can take more than a second) and state synchronization (which takes about 13ms) are bottlenecks during failure recovery.

Our proposal: FTMesh aims to reduce client re-connection time by having the virtual network transparently reroute traffic to the target instance; and reduce state synchronization time by proactively replicating state to the target instance before failure. FTMesh leverages service mesh [2] and its network abstractions [6] for both, and thus provides a general mechanism that can be used by any interactive application.

2 Existing Approaches

Microservice orchestrators, e.g., Kubernetes [4] and Nomad [5], restart services on failures. This approach assumes that services are either stateless, or can reconstruct any state lost on failure by reading an application generated checkpoint from stable storage. Interactive applications often employ services that are not stateless, and their performance requirements can make periodic checkpoints infeasible. Prior work has suggested alternate approaches where the orchestrator checkpoints containers [10, 11] using lightweight mechanisms, but as we showed above, state synchronization and connection re-establishment latencies make these approaches unsuitable for interactive applications.

Low-overhead fault tolerance approaches have formerly been developed for network functions [7–9] whose processing requirements are similar to those of interactive applications. However, network functions have simpler semantics, state access patterns, and communication graphs than the microservice based applications we target. Our work is thus inspired by these prior efforts, but also distinct from them.

3 Design

FTMesh functionality is implemented using sidecars, which are per-instance proxies [1] in the service mesh that have visibility into all traffic sent or received by the instance.

State Synchronization: Figure 2a shows FTMesh's state synchronization mechanism, which runs during normal application execution. Our mechanism assumes that services identify critical state (that required for recovery), and includes updated values for this state in the request and response bodies. We do not require changes to the RPC library to do so, since we make the proxy to remove this data before messages are delivered to the RPC library. This state is replicated at FTMesh agents in the target service's proxy. As an optimization, we can allow state synchronization to be piggybacked on existing application traffic.

Recovery and Re-routing: When failures are detected, the FTMesh recovery process proceeds as follows: (a) first, the target's FTMesh agent uses the synchronized state to recover sessions of the failed service; and (b) once these new sessions has been initialized, we update the service mesh network to forward traffic intended for the failed instance to the new instance. We show this in Figure 2b.

Optimization: Pushing the connection information to the whole cluster wastes communication resources on unrelated instances, and cannot ensure critical connections get recovered first. We thus rely on existing Service Mesh controller's capability of broadcasting configurations to non-critical instances eventually (slow update).

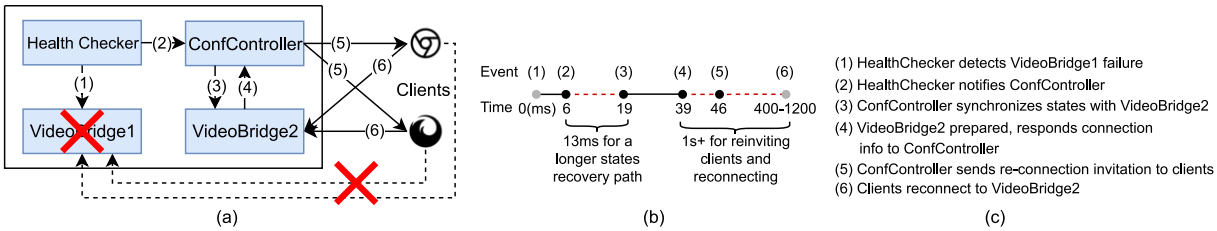
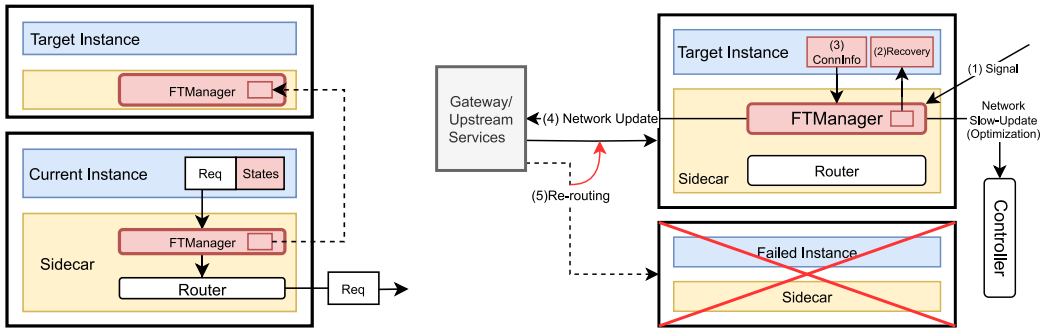


Figure 1. Failover procedure and time breakdown of Jitsi.



(a) The state synchronization procedure. In reality, the state is synchronized with multiple target instances.

(b) The failover procedure.

Figure 2. The design of FTMesh.

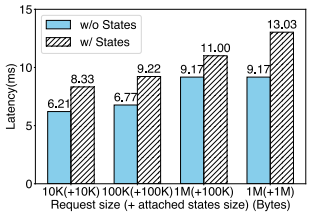


Figure 3. Overheads of state synchronization during normal requests.

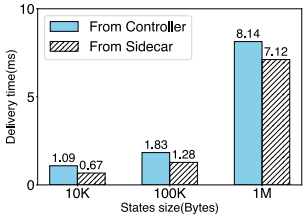


Figure 4. Performance of state recovery.

4 Preliminary Results

We build a prototype of FTMesh and do micro-benchmarks on an LAN cluster.

Operation Overhead: We measure the time of requests of different size settings. Figure 3 shows that FTMesh incurs 20% latency overhead when attaching 100KB state to a 1MB request, and increases average latency by 2.57ms, 32%.

States Recovery: We measure the time of delivering the state to an instance from a local Sidecar or a remote Controller. Figure 4 presents the time difference. FTMesh can achieve 27% shorter delivery time on average.

Re-routing: We evaluate the re-routing performance by measuring the time from new routing configuration being composed to data arriving at the new instance. Our prototype re-routes UDP traffics with 50.43ms on average.

5 Challenges

#1 Non-determinism: Many services are multi-threaded and have global state for higher throughput. Concurrent nature requires correct replication and recovery procedures to avoid messing up service state.

#2 Application integration: We rely on service to attach state to the traffic. Assisting existing services to identify critical state, and providing interfaces for building new applications require careful design. This is rather critical as we assist synchronizing state under multi-thread contention.

#3 Growing state size and complexity: Modern services contain large state and complicated memory references. How to ensure the performance as the state size and (un)marshalling time grow challenges state update and encoding mechanisms.

References

[1] 2023. Envoy Proxy. https://www.envoyproxy.io/

[2] 2023. Istio. https://istio.io/latest/

[3] 2023. Jitsi. https://jitsi.org/

[4] 2023. Kubernetes. https://kubernetes.io/

[5] 2023. Nomad. https://developer.hashicorp.com/nomad

[6] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 229–236. https://doi.org/10.1145/3484266.3487379

[7] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. 2020. Fault Tolerant Service Function Chaining. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 198–210. https://doi.org/10.1145/3387514.3405863 event-place: Virtual Event, USA.

[8] Sameer G Kulkarni, Guyue Liu, K. K. Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. REINFORCE: Achieving Efficient Failure Resiliency for Network Function Virtualization Based Services. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3281411.3281441 event-place: Heraklion, Greece.

[9] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 227–240. https://doi.org/10.1145/2785956.2787501 event-place: London, United Kingdom.

[10] Diyu Zhou and Yuval Tamir. 2020. Fault-Tolerant Containers Using NiLiCon. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1082–1091. https://doi.org/10.1109/IPDPS47924.2020.00114 ISSN: 1530-2075.

[11] Diyu Zhou and Yuval Tamir. 2022. RRC: Responsive Replicated Containers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 85–100. https://www.usenix.org/conference/atc22/presentation/zhou-diyu