APPENDIX A
CLIENT-DRIVEN HYPERLEDGER FABRIC SERVICE

Appendix A will introduce the client-driven Hyperledger Fabric service.

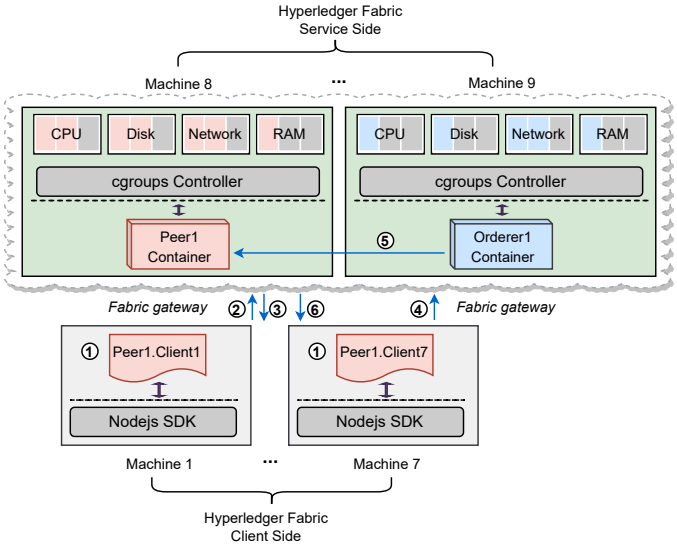


Fig. 1. A sample of client-driven Hyperledger Fabric is shown, where, to make it simple, we consider one peer (Peer1) and one ordering service node (i.e., Orderer1). The workload generator (i.e., the client-side) is based on the Hyperledger Fabric SDK for Node.js, which is deployed on seven machines, each with a unique client membership of Peer1 (i.e., Peer1.Client1, Peer1.Client2, $\cdots$, Peer1.Client7, respectively). Consider that there is one channel where Peer1 is an endorsing peer. Specifically, in step ①, clients of Peer1 initiate seven transaction proposals. In step ②, clients of Peer1 concurrently send these seven transaction proposals to Peer1 for simulation and endorsement. In step ③, Peer1 simulates and endorses these seven transaction proposals and passes back the proposal response to the corresponding client. In step ④, each client assembles the corresponding proposal response, prepares a transaction, and then forwards the transaction to the ordering service concurrently. In step ⑤, a new block created/cut from the ordering service is delivered to Peer1 for validation and commitment. Finally, all clients will be notified of the committing result in step ⑥.

There are two main parts: the Hyperledger Fabric service and the client. The Hyperledger Fabric service is the platform that hosts the core components of Hyperledger Fabric, including peers, orderers, and smart contracts. The client-side only hosts membership files generated by the certificate authority (CA) and runs the Node.js SDK Toolkit to interact with the Hyperledger Fabric service. It means there is no core component of Hyperledger Fabric on the client side, except for some application programming interfaces (APIs) exposed by the service side. Additionally, there is a bridge, known as the fabric gateway, connecting the two main parts. The fabric gateway interacts with the Hyperledger Fabric service, such as peers, orderers, and CAs, on behalf of the client side, and exposes a simple gRPC interface between the client and the service sides. It avoids strain on the service side caused by direct access from the client side.

Fig. 1 shows a sample of client-driven Hyperledger Fabric. In step ①, the client initializes a transaction proposal. In step ②, the client submits the transaction proposal to the endorsing peer in the execute phase. In step ③, the endorsing peer in the execute phase verifies the transaction proposal and takes the transaction proposal as an input argument to invoke the user chaincode to get the target transaction result, i.e., a response value and a read-write set. Then, the endorsing peer in the execute phase passes the transaction result and its own signature to the client. Invoking a transaction in the execute phase is also referred to as simulation, as no updates are made to the ledger during this phase. In step ④, upon receiving the proposal response from the execute phase, the client assembles endorsements into a transaction and submits the transaction to the ordering service. In step ⑤, the ordering service is mainly responsible for ordering transactions sequentially and cutting a new block without inspecting the entire content of each transaction. Once the new block is received from the order phase, the validate phase validates and commits the new block. In step ⑥, a peer emits an event per transaction to notify the corresponding client.

APPENDIX B
RAFT-BASED ORDERING SERVICE

Appendix B will introduce the Raft-based ordering service.

Hyperledger Fabric utilizes the Raft-based ordering service to address the classic consensus challenge in blockchain technology. The Raft protocol is based on the Quorum mechanism. It means that a Raft consensus is equivalent to an agreement among a Quorum. In the Raft-based ordering service, there is at most one OSN leader at any time, while the others become OSN followers. The OSN leader is the node that maintains the most recent committed state at any time. The relationship between the OSN leader and followers is that the OSN leader can decide on a value only if an agreement on the value is achieved among at least $\lceil \frac{k}{2} \rceil$ OSNs, where k is the total number of the OSNs, including the OSN leader. Upon the OSN leader deciding on a value, it will be replicated concurrently to all OSN followers over the network. The Quorum-based decision-making method brings some communication bandwidth cost between the OSN leader and followers. It also gains the property of crash fault-tolerant (CFT) and provides a high-availability ordering service.

A Raft-based ordering service consists of multiple Raft-based OSNs. Each Raft-based OSN has three core modules: the consensus module, the replicated write-ahead logging (WAL) module, and the replicated state machine module. Specifically, the consensus module is responsible for receiving client requests from the network, encapsulating these requests into log entries, and replicating the log entries to the replicated WAL module among OSN followers over the network. Upon receiving the log entries from the consensus module, the replicated WAL module stores the log entries in its local memory and responds to the consensus module. The Raft consensus module enforces a property of log matching to ensure that the log entry sequences among a majority of Raft-based OSNs are consistent. Additionally, a crucial property of the replicated state machine module is that, given a well-sequenced input of log entries, the output of the replicated state machine preserves the same sequence. Therefore, upon receiving the replies to the sequenced log entry from the OSN followers, the OSN leader can now safely feed the sequenced

log entries to the replicated state machine module, apply the state, and create a new block.

There are two assumptions: first, we consider no frequent three-handshakes between any two OSNs. We used Wireshark version 3.4.5 (*v3.4.5-0-g7db1feb42ce9*) to capture the network packets between the end-to-end OSNs and found that the end-to-end OSNs use the TCP protocol for data transmission, with only a three-handshake between any two OSNs. We do not need to consider the extra cost of frequent three-handshakes since the first three-handshake was established during the channel configuration in Hyperledger Fabric. Therefore, we assume that the end-to-end connection between any two OSNs is always in a TCP connection in a reliable cluster network. Second, we consider no CPU-intensive workload in the order phase of Hyperledger Fabric. The job of the ordering service is to sequence the transactions from the clients over the network without making any judgment of transaction content [1]. Therefore, we consider that there are sufficient CPU time slots to allocate for ordering transactions during the order phase.

APPENDIX C
BLOCK CUTTER AND IDLE TIME

Appendix C will introduce the block cutter and the corresponding idle time.

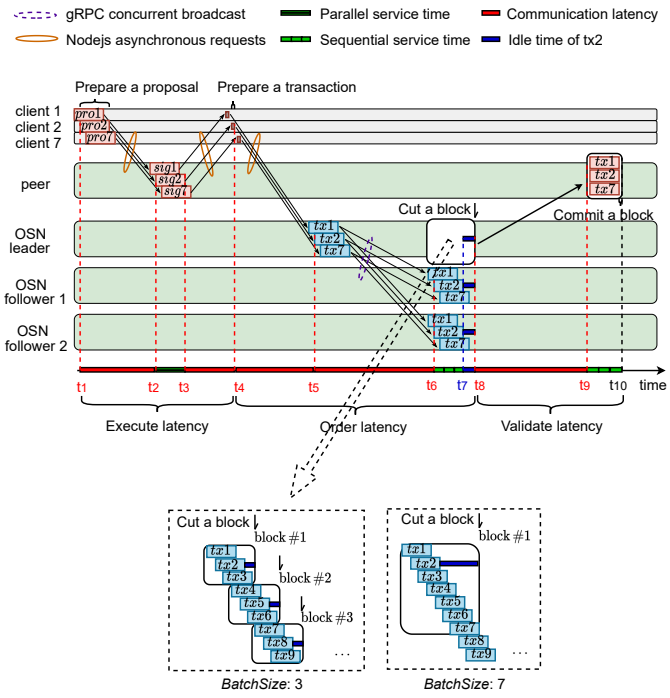


Fig. 2. A simple case of the block cutter and the corresponding idle time in the Raft-based ordering service with a *BatchSize* of 3 (or 7).

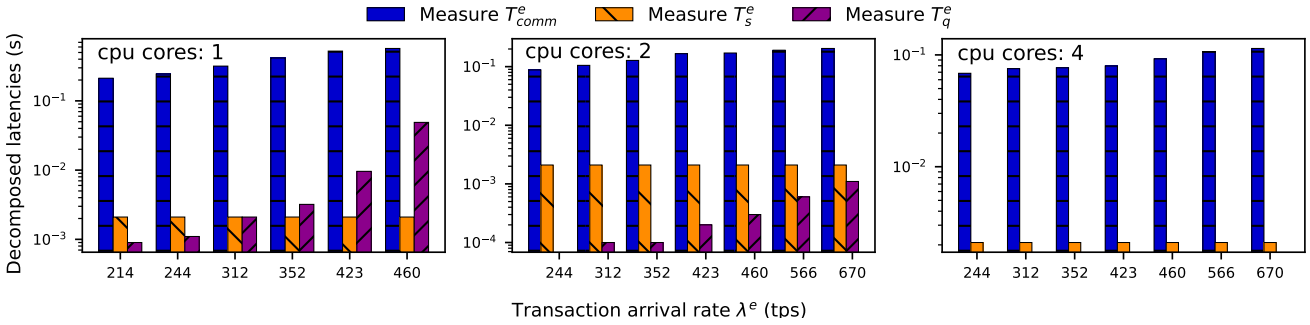
We find that the block cutter adds the idle time to a transaction. The Raft consensus protocol handles data transaction by transaction, while the Raft-based ordering service handles data block by block. In other words, a transaction that quickly achieves a Raft consensus may not be included in a new block immediately due to different granularity levels. It means that it needs to wait until enough transactions arrive and a new block is created. Therefore, the idle time of a transaction when batching a new block is introduced by the block cutter on the OSN leader. Two key parameters determine the idle time: *BatchTimeout* and *BatchSize*. The *BatchTimeout* determines the time to wait until creating a new block, and the *BatchSize* determines the total number of transactions and the total transaction size of a new block. The ordering service generates a new block if either of the configuration parameters is satisfied.

Fig. 2 shows a simple case of the block cutter and the idle time in the Raft-based ordering service with a *BatchSize* of 3. It also illustrates a different scenario with the block cutter and idle time, using a *BatchSize* of 7. Specifically, in the order phase, the client submits transaction $tx2$ to the OSN leader at timestamp t_4. The OSN leader receives the transaction at timestamp t_5 and then broadcasts the transaction $tx2$ to $k-1$ OSN followers. And a Raft consensus among k OSNs on transaction $tx2$ is achieved at timestamp t_6. But the condition of *BatchSize* is not satisfied yet, and transaction $tx2$ waits for $t_8 - t_7$ seconds until the *BatchSize* condition is satisfied and a new block is created by the OSN leader at timestamp t_8. Therefore, the idle time of transaction $tx2$ is $t_8 - t_7$.

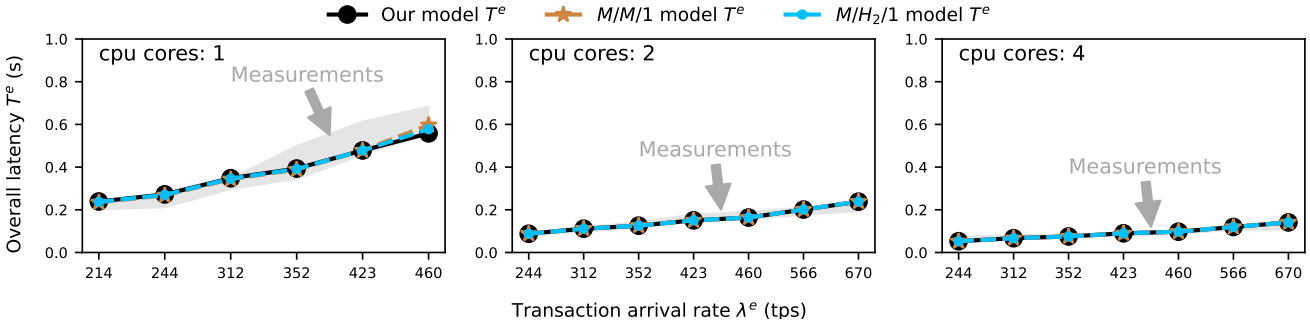
APPENDIX D
VALIDATION RESULTS

Appendix D will show the validation results in two different clusters.

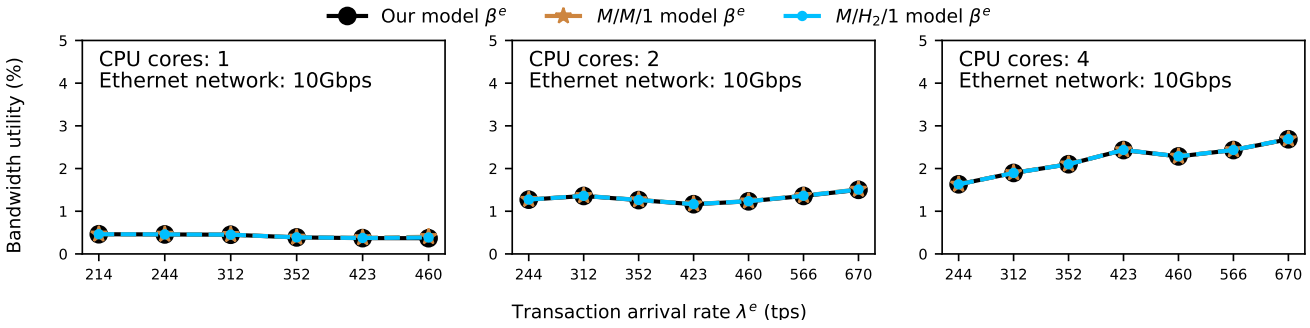
[1]The Raft-based ordering service in Hyperledger Fabric 2.2 LTS: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html



(a) The effects of cpu cores on decomposed latencies T^e_{comm}, T^e_s, and T^e_q in the execute phase

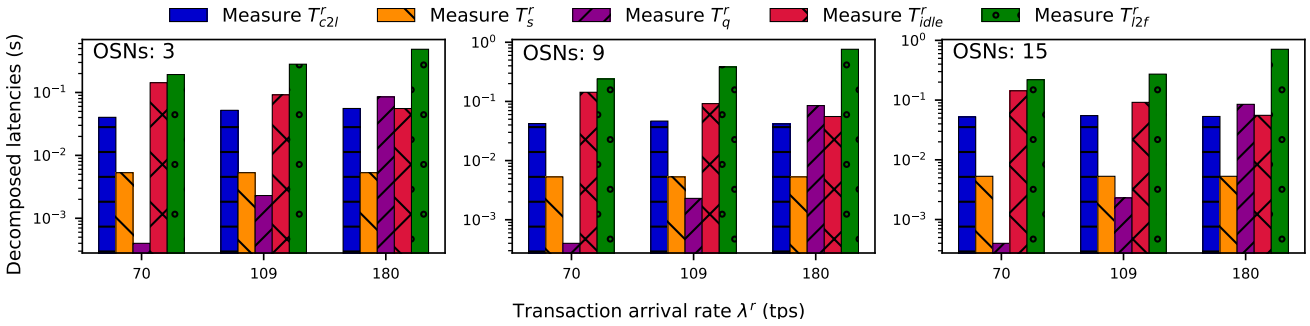


(b) The effects of cpu cores on latency T^e in the execute phase

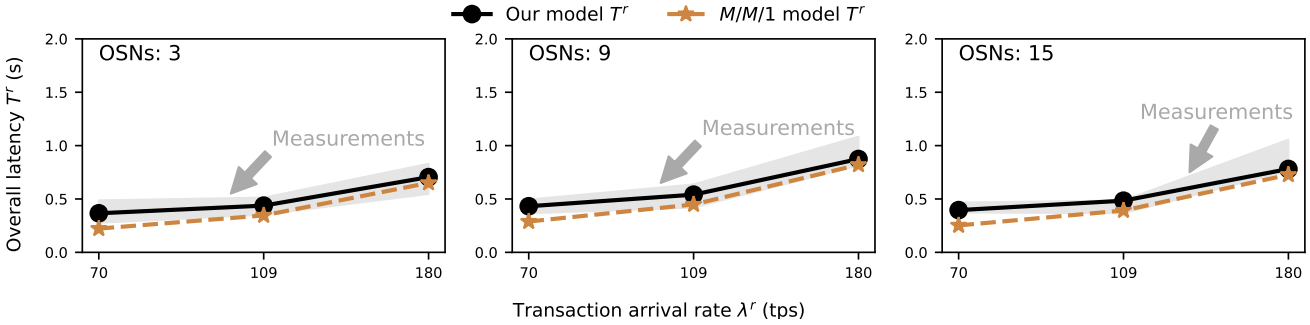


(c) The effects of cpu cores on bandwidth utilization β^e in the execute phase

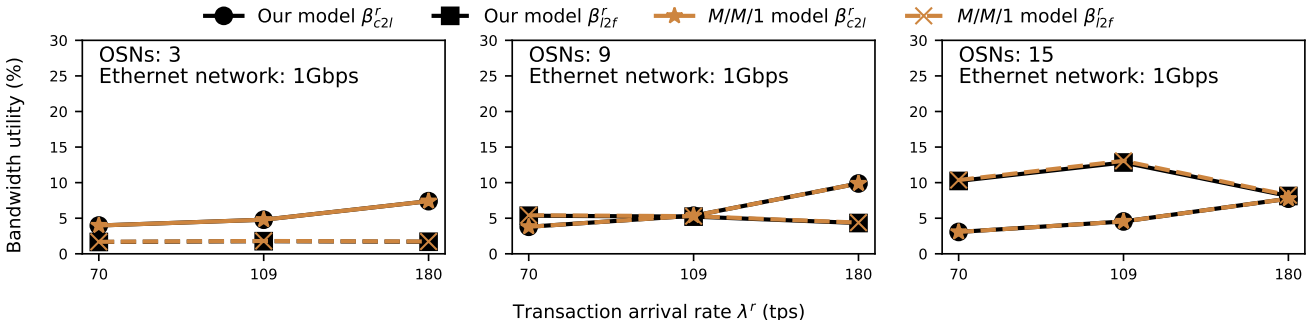
Fig. 3. The effects of CPU cores on the latency and bandwidth utilization in the execute phase. There is an endorsing peer with $c = 1, 2, 4$ CPU core(s), respectively. The remaining machines comprise seven clients and three ordering service nodes. All machines are connected via a bi-directional non-blocking switch to an Ethernet network with a speed of 10 Gbps. Remark that $T^e = T^e_{comm} + T^e_s + T^e_q$.



(a) The effects of OSNs on decomposed latencies T^r_{c2l}, T^r_{l2f}, T^r_{idle}, T^r_s, and T^r_q in the order phase

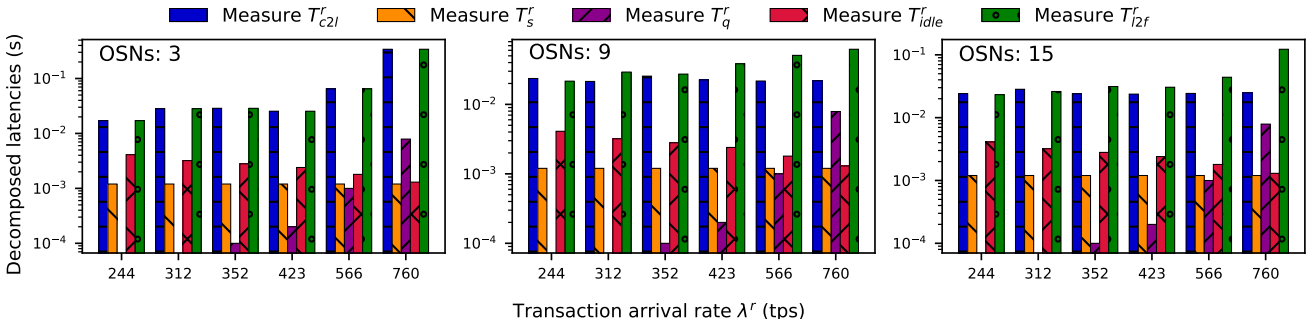


(b) The effects of OSNs on latency T^r in the order phase

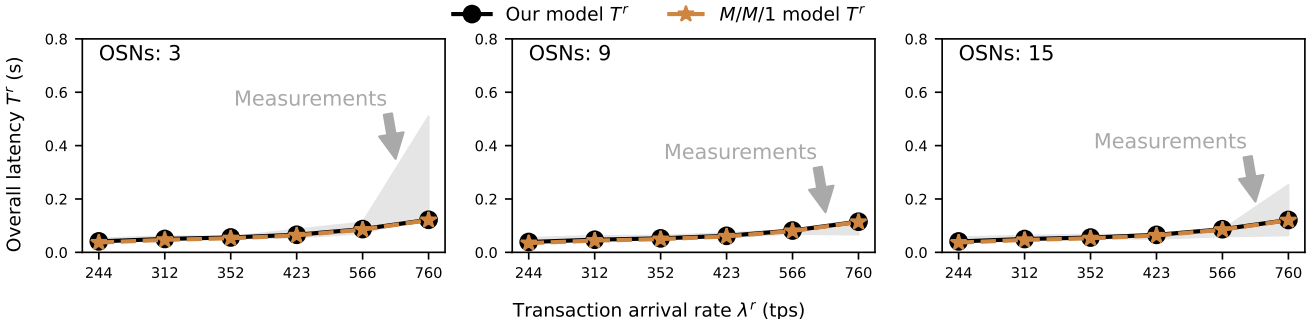


(c) The effects of OSNs on bandwidth utilization in the order phase

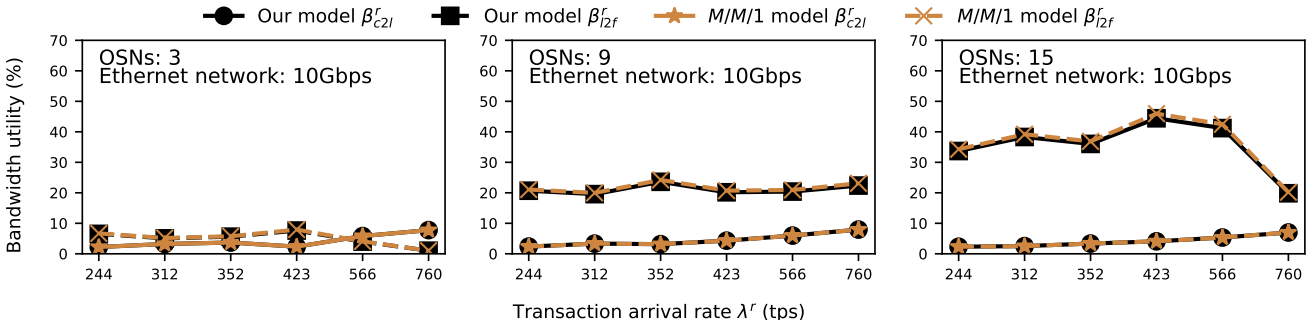
Fig. 4. The effects of OSNs on the latency and bandwidth utilization in the order phase. There is an ordering service with $k = 3, 9, 15$ OSNs, respectively. The *BatchSize* is 20 and the *BatchTimeout* is 1. The rest of the machines are seven clients and one peer. All machines are connected via a bi-directional non-blocking switch to an Ethernet network with a speed of 1 Gbps. Remark that $T^r = T^r_{c2l} + T^r_{l2f} + T^r_{idle} + T^r_s + T^e_q$.



(a) The effects of OSNs on decomposed latencies T^r_{c2l}, T^r_{l2f}, T^r_{idle}, T^r_s, and T^r_q in the order phase

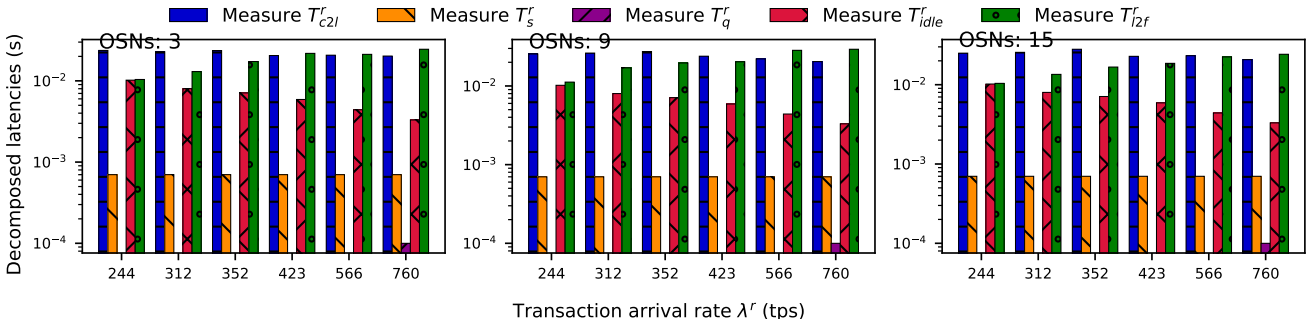


(b) The effects of OSNs on latency T^r in the order phase

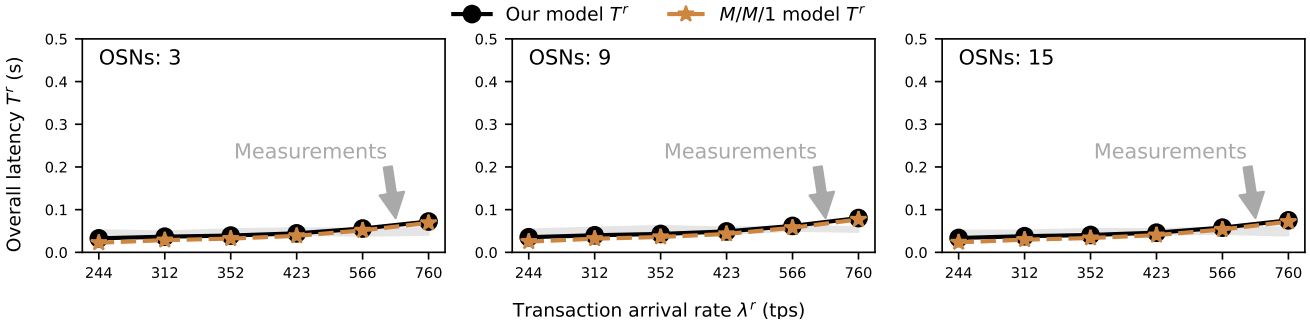


(c) The effects of OSNs on bandwidth utilization in the order phase

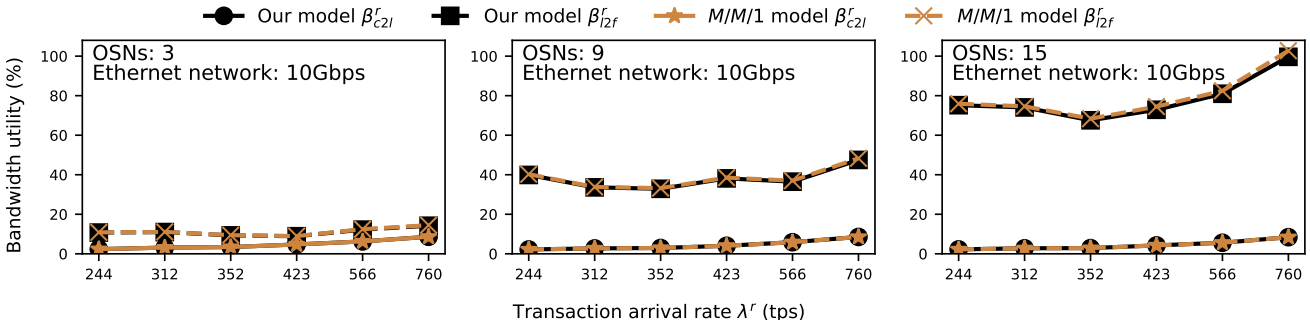
Fig. 5. The effects of OSNs on the latency and bandwidth utilization in the order phase. There is an ordering service with $k = 3, 9, 15$ OSNs, respectively. The *BatchSize* is 50 and the *BatchTimeout* is 1. The rest of the machines are seven clients and one peer. All machines are connected via a bi-directional non-blocking switch to an Ethernet network with a speed of 10 Gbps. Remark that $T^r = T^r_{c2l} + T^r_{l2f} + T^r_{idle} + T^r_s + T^e_q$.



(a) The effects of OSNs on decomposed latencies T^r_{c2l}, T^r_{l2f}, T^r_{idle}, T^r_s, and T^r_q in the order phase

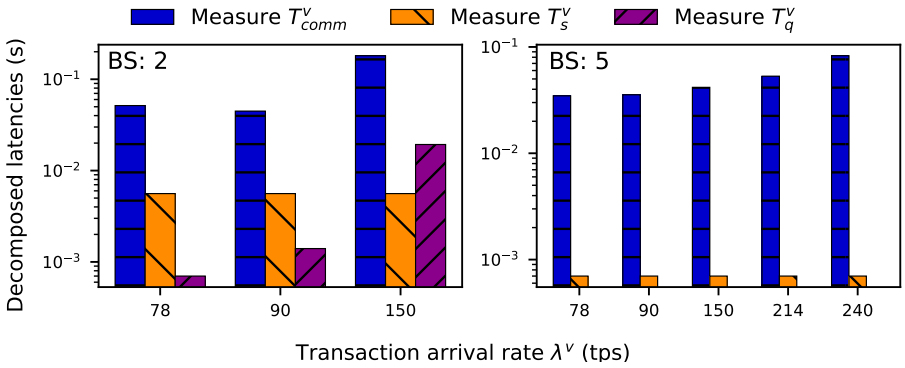


(b) The effects of OSNs on latency T^r in the order phase

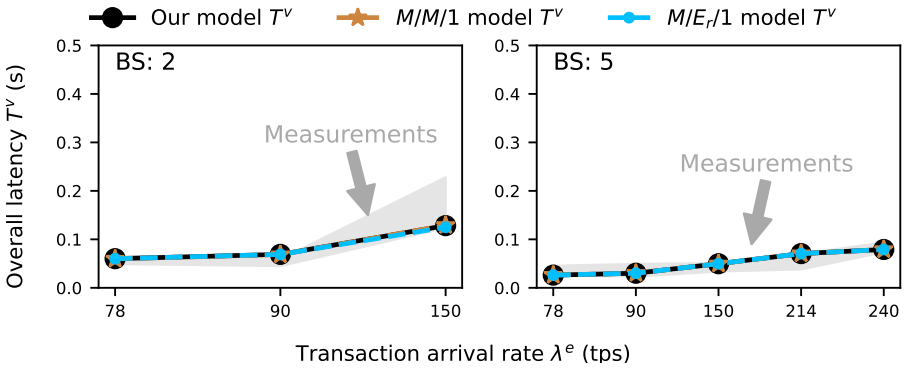


(c) The effects of OSNs on bandwidth utilization in the order phase

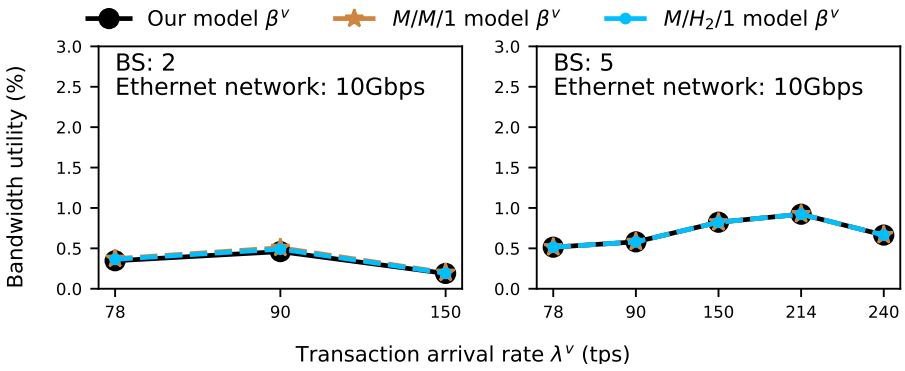
Fig. 6. The effects of OSNs on the latency and bandwidth utilization in the order phase. There is an ordering service with $k = 3, 9, 15$ OSNs, respectively. The *BatchSize* is 5 and the *BatchTimeout* is 1. The remaining machines comprise seven clients and one peer. All machines are connected via a bi-directional non-blocking switch with a 10 Gbps Ethernet network. Remark that $T^r = T^r_{c2l} + T^r_{l2f} + T^r_{idle} + T^r_s + T^e_q$.



(a) The effects of storage devices on decomposed latencies T^v_{comm}, T^v_s, and T^v_q in the validate phase



(b) The effects of storage devices on latency T^v in the validate phase



(c) The effects of storage devices on bandwidth utilization in the validate phase

Fig. 7. The effects of storage devices on the latency and bandwidth utilization in the validate phase. A committing peer has an SSD storage device. The rest of the machines are seven clients and three ordering service nodes. All machines are connected via a bi-directional non-blocking switch with a 10 Gbps Ethernet network. Remark that $T^v = T^v_{comm} + T^v_s + T^v_q$.