## APPENDIX A
### CLIENT-DRIVEN HYPERLEDGER FABRIC SERVICE

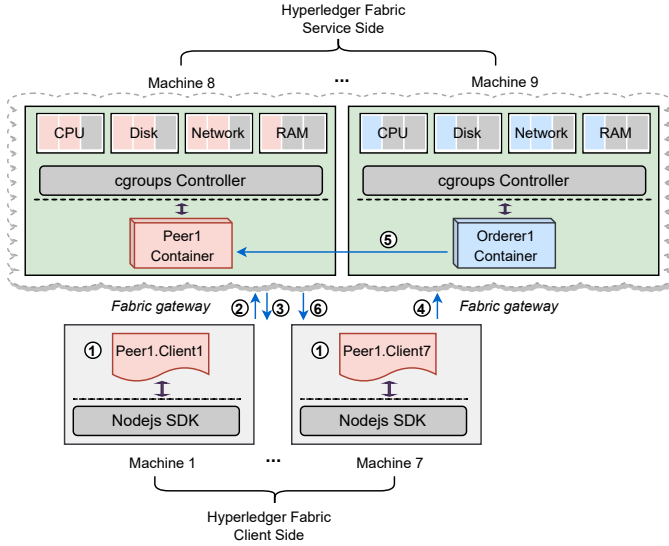Appendix A introduces the client-driven Hyperledger Fabric service.



Fig. 1. A sample of the client-driven Hyperledger Fabric service. There is one service and one client. The service includes one peer, one ordering service node, and one channel. The client is based on the Hyperledger Fabric SDK for Node.js, deployed on seven machines.

Hyperledger Fabric can be divided into two parts: Hyperledger Fabric service and Hyperledger Fabric client. The Hyperledger Fabric service is the platform that hosts the core components of Hyperledger Fabric, including peers, orderers, and smart contracts. The Hyperledger Fabric client 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 these two 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 the client-driven Hyperledger Fabric service. 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, since 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 responsible for ordering transactions sequentially and cutting a new block without inspecting the entire content of each transaction. Once a 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 for each transaction to notify the corresponding client.

## APPENDIX B
### RAFT-BASED ORDERING SERVICE

Appendix B describes the Raft-based ordering service.

Hyperledger Fabric utilizes the Raft-based ordering service to address the consensus challenge in blockchain technology. The Raft protocol is based on a Quorum mechanism, meaning 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 a majority of OSNs. Upon the leader deciding on a value, it will be replicated concurrently to all OSN followers over the network. Such a Quorum-based decision-making method gains the property of Crash Fault-Tolerant (CFT) and provides a high-availability ordering service. It also brings some bandwidth costs between the OSN leader and followers.

A Raft-based ordering service consists of several 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 one important property called *log matching* to ensure that the log entry sequences among a majority of Raft-based OSNs are consistent. Another important property is that given a well-sequenced input of log entries, the output of the replicated state machine preserves the same sequence. As a result, upon receiving the replies to the sequenced log entry from the OSN followers, the OSN leader can now consistently and safely feed the sequenced log entries to the replicated state machine module, apply the state, and create a new block.

There are two assumptions in the Raft-based ordering service. The first assumption is to 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. It means that it does not need to consider the extra cost of frequent

three-handshakes because the first three-handshake was established during the channel configuration in Hyperledger Fabric. Therefore, it is reasonable to assume that the end-to-end connection between any two OSNs is always in a TCP connection in a reliable cluster network. The second assumption is to 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, it is reasonable to consider that there are sufficient CPU time slots to allocate for ordering transactions during the order phase.

## APPENDIX C
### BLOCK CUTTER AND WAITING TIME

Appendix C introduces the block cutter and the corresponding waiting time.
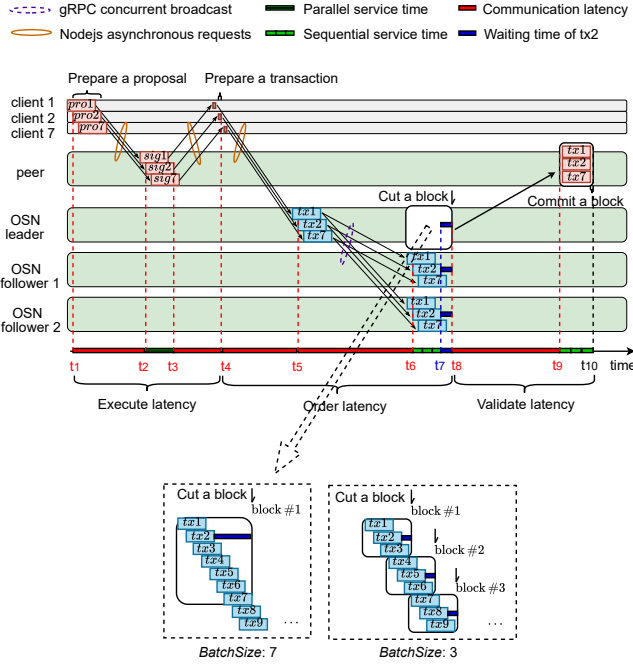


Fig. 2. A simple illustration of the block cutter and the corresponding waiting time in the Raft-based ordering service with a *BatchSize* of 7 (or 3).

The block cutter brings a waiting time to a transaction. Specifically, the Raft consensus protocol handles data from a transaction level, while the Raft-based ordering service handles data from a block level. It means that a transaction that quickly achieves a Raft consensus may not be included in a new block immediately. Because the block cutter needs to wait until enough transactions arrive and a new block can be created then. Therefore, the waiting time of a transaction is introduced by the block cutter on the OSN leader when batching a new block.

[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
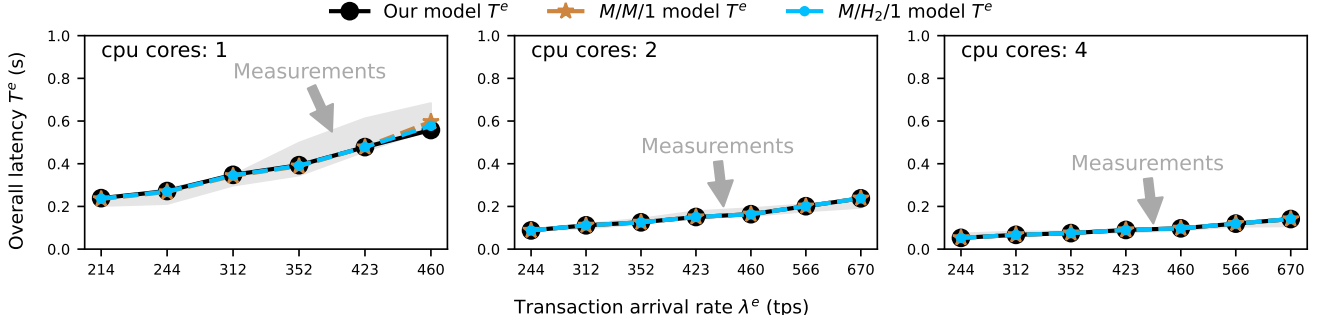
There are two block-setting parameters that determine the waiting time: *BatchTimeout* and *BatchSize*. The *BatchTimeout* determines the time to wait until creating a new block, and the *BatchSize* determines the total number of transactions and the total transaction size to wait until creating a new block. The ordering service generates a new block if either of the two block-setting parameters is satisfied.

Fig. 2 shows a simple illustration of the block cutter and the waiting time in the Raft-based ordering service with a *BatchSize* of 3 and 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 OSN followers. And a Raft consensus among OSNs on transaction $tx2$ is achieved at timestamp $t_6$. Since 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 waiting time of transaction $tx2$ is $t_8 - t_7$.

## APPENDIX D
### VALIDATION RESULTS

Appendix D illustrates the validation results in cluster two.

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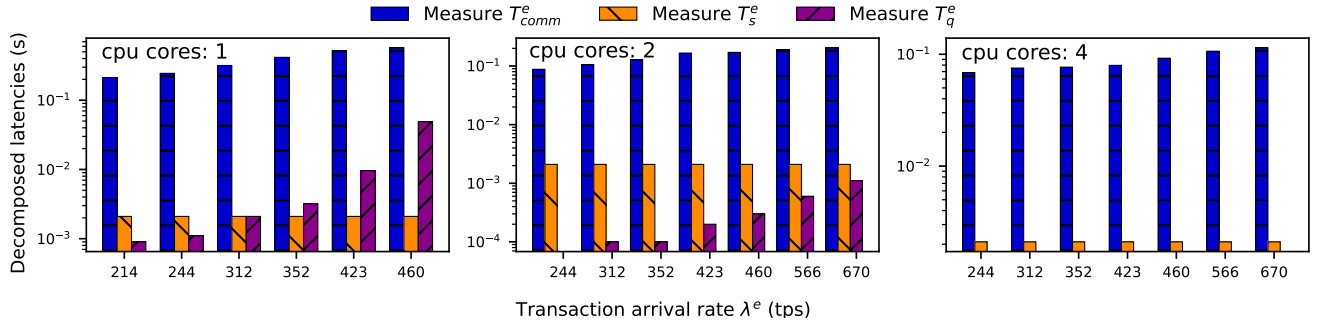


(b) The effects of cpu cores on decomposed latencies $T^e_{comm}$, $T^e_s$, and $T^e_q$ in the execute phase, where $T^e = T^e_{comm} + T^e_s + T^e_q$
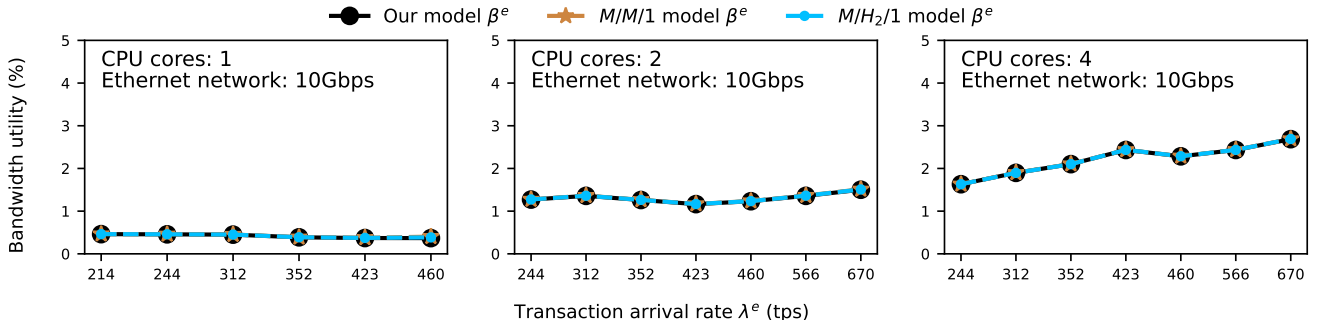


(c) The effects of cpu cores on bandwidth utilization $\beta^e$ in the execute phase

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.
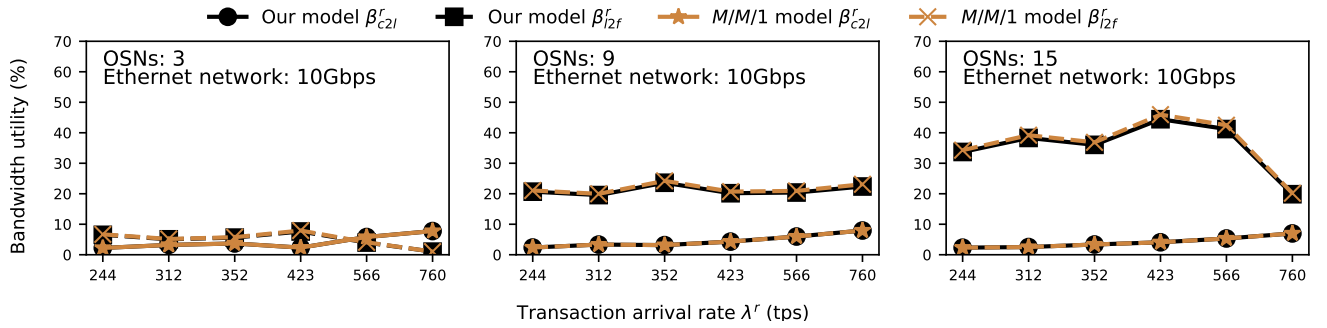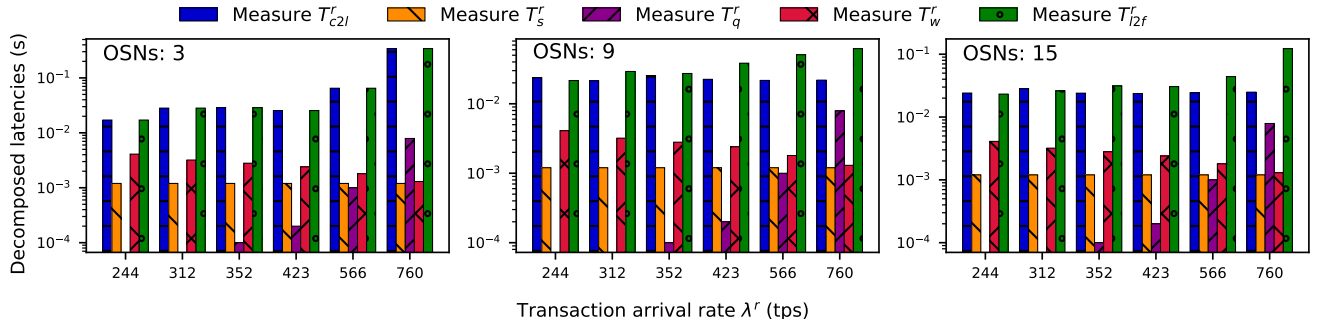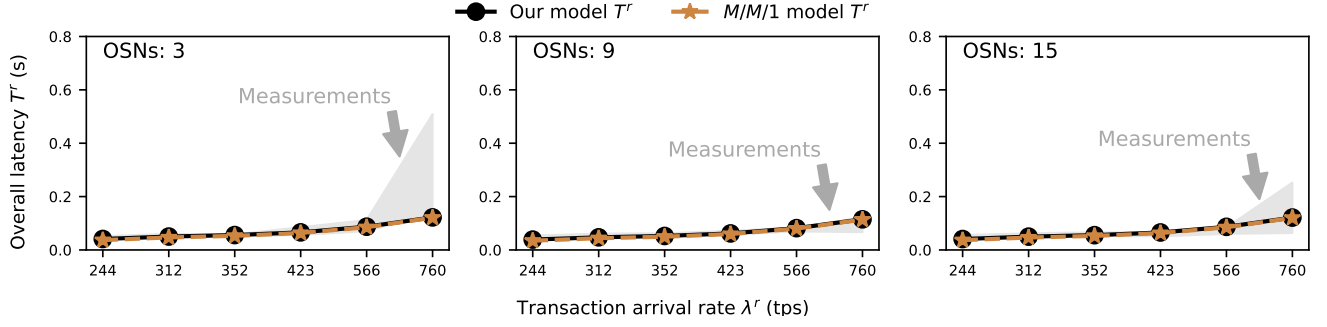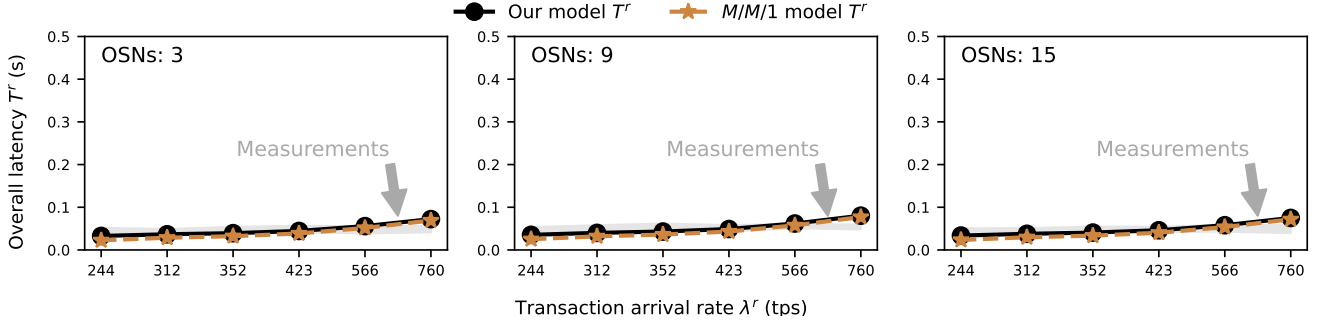
(a) The effects of OSNs on latency $T^r$ in the order phase



(b) The effects of OSNs on decomposed latencies $T^r_{c2l}$, $T^r_{l2f}$, $T^r_w$, $T^r_s$, and $T^r_q$ in the order phase, where $T^r = T^r_{c2l} + T^r_{l2f} + T^r_w + T^r_s + T^e_q$



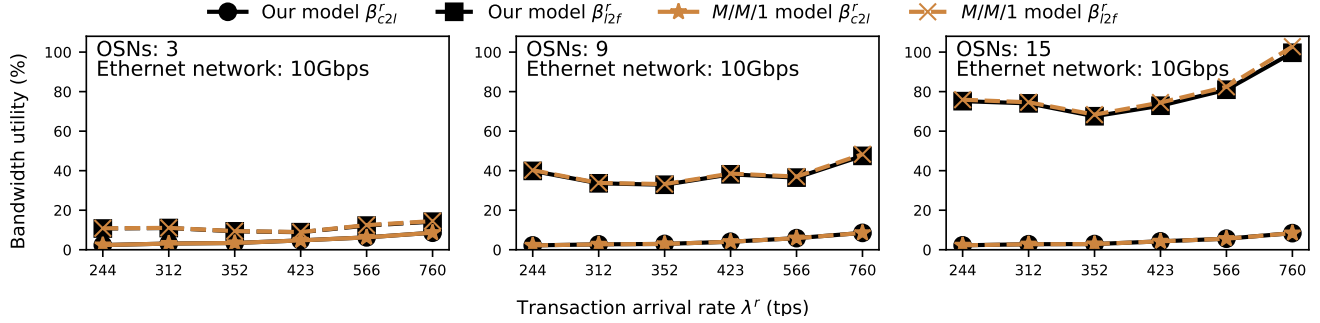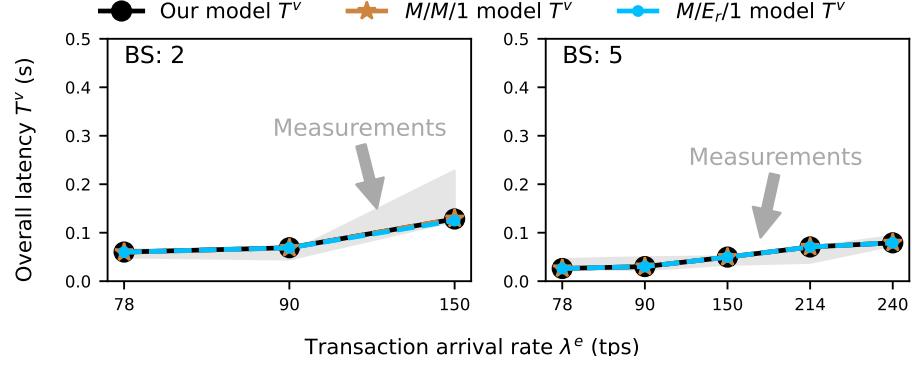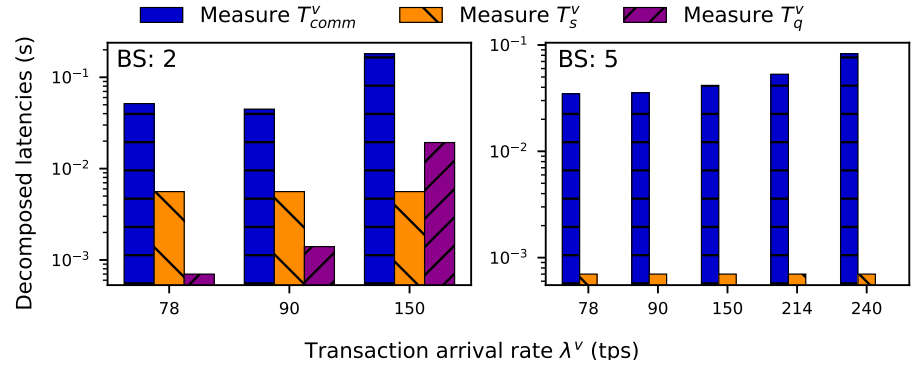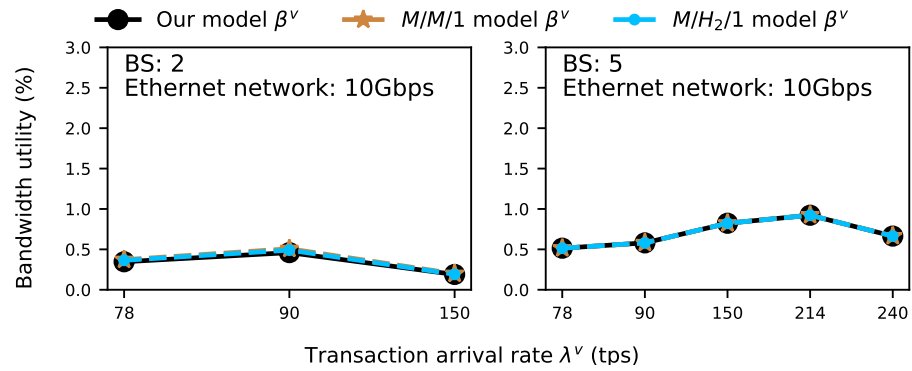(c) The effects of OSNs on bandwidth utilization in the order phase

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 2 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.

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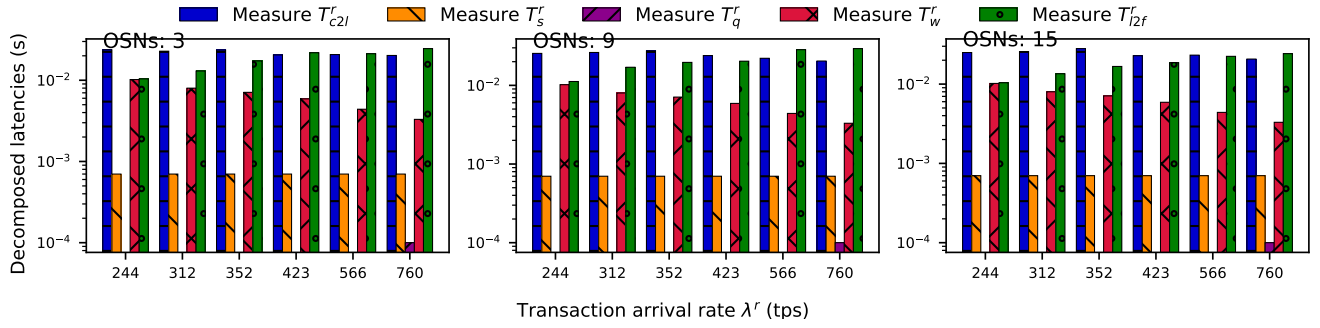


(b) The effects of OSNs on decomposed latencies $T^r_{c2l}$, $T^r_{l2f}$, $T^r_w$, $T^r_s$, and $T^r_q$ in the order phase, where $T^r = T^r_{c2l} + T^r_{l2f} + T^r_w + T^r_s + T^e_q$



(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 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.

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



(b) The effects of storage devices on decomposed latencies $T^v_{comm}$, $T^v_s$, and $T^v_q$ in the validate phase, where $T^v = T^v_{comm} + T^v_s + T^v_q$



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

Fig. 6. 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.