factoryOS - A Distributed and Self-Organizing Planning System in a Supply-Chain Context

Department of Computer Science
Stanford University

andrej@stanford.edu
cchuanqi@cs.stanford.edu
nishantr@cs.stanford.edu
raok@stanford.edu

Abstract

Supply-Chain management systems deal with massive amounts of data. In today's scenario, with numerous inventories to track, they're unable to work efficiently because 1. they are not fault-tolerant 2. their data is stored on a single database, and 3. they require manual reconfiguration in case of errors. We introduce `factoryOS`, a distributed and self-organizing planning system that can cope with these challenges. `factoryOS` is robust to network delays and fault-tolerant, partitions inventory data natively, and works without any manual intervention. Our novel design point is that we're able to guarantee uninterrupted production even with multiple failures. With `factoryOS`, partition shards are created natively and nodes handle their own processes individually. Our experimental results show that `factoryOS` can be easily scaled to a large number of nodes and is fault-tolerant to a high degree of failures. Code and scripts are available at FactoryOS.

1 Introduction

Given the trend of globalization, manufacturing processes are becoming more and more complex [1]. For example, to produce one car, 30,000 individual parts need to be assembled. Most of those parts are sourced from thousands of different third-party providers. A delay in just one section of the supply chain can slow down the production and distribution of critical components [2]. This issue is exacerbated by the fact that complex supply chains are extremely sensitive to external political, economic, and environmental factors [3]. Also, consumers as well as producers demand personalization of their purchased products, which makes the production of goods even more complicated [4] due to rapidly changing requirements.

Therefore, managing every product's supply chain can be a challenging task. Currently, most companies centrally maintain and manage their own IT systems for supply chain management [5]. That leads to three challenges. First, supply chain systems need to handle cases when requirements change for a product or when a node in the supply chain goes down. The status quo is to manually reconfigure the transactions between different nodes or oversupply on specific parts to prevent a slow-down. Second, current systems are usually not fault-tolerant. If one node goes down, the whole system is stopped until the issue is found and fixed manually [6]. Third, many companies store all their supply chain data in one single database with terabytes of data. This leads to bottlenecks with high frequency database queries or high locking contention on the data [7].

In this paper, we aim to address all these three challenges with our distributed and self-organizing supply-chain manufacturing system, `factoryOS`. `factoryOS` guarantees four system properties:

CS 244b Distributed Systems, Spring, 2020, California, USA.

1. **Robustness to network delays:** The system is able to continue manufacturing even if there are network delays.
2. **Fault-Tolerance to node crashes:** The system is fault-tolerant by making nodes only depend on their neighbors. Even if the leader node crashes, nodes continue production while a new leader is being elected.
3. **Native Inventory Partitioning:** Instead of having a large database of inventory history, each individual node manages its own inventory data. Also, there is no need for a clever algorithm to partition shards; the shards are created natively.
4. **Uninterrupted Production without manual intervention:** `factoryOS` dynamically finds viable flows in case a node crashes or if requirements for a node change.

2 Related Work

Supply Chain Networks are often modeled and analyzed using Petri nets and we use simplified version of Petri Nets which can be extended with more stochastic characteristics of supply chains[8] [9]. A multi-agent based approach is proposed to enable manufacturing systems to make fast and frequent reconfiguration of the production systems [10]. Latest work in this area is Token-flow Supply Chains: the Colored Petri-nets model of digital supply chain resolves the limitation of Petri-nets to achieve the distinctness of markings, and "Distributed Leger System" manages all supply chain transactions and digital events. With DLSs, each transactions are verified by a consensus of the DLSs' participating nodes, once the consensus is reached and stored in all participating nodes, thus it solves the limitation of single point of failure of centralized leger system. In contrast, our distributed system automatically elects a central leader. This leader makes optimal decisions of how to reconfigure the supply flow globally in case a node fails.

There are similarities between our system and Apache Kafka [11]. Both systems are structured in clusters with producer and consumer APIs. However, in Kafka's case, the sender can send messages to Kafka, while the recipient gets messages from the stream published by Kafka. Kafka acts as a central distributor. In our case, sender and recipients communicate directly with each other and our system only steps in when the current flow needs to be changed.

3 Proposed Approach

3.1 Overview

This section covers the design of `factoryOS` and its components. A production system in `factoryOS` consists of a set of nodes. We assume the existence of a fixed set of item types. Each node represents a stage in the system consisting of the possible consumed and produced items within the supply chain. There can be different producers of the same item type in our system, this is key in order to support uninterrupted flow of production. Each node has a certain node-ID and needs certain items as input requirements to be able to produce a resulting item. This dependency between two nodes can be represented as an edge in the DAG. During initialization, we reach consensus on a possible supply-chain flow. The agreed upon supply-chain flow can also be represented as a DAG. We'll go over each component in detail in the following sections.

An item requirement consists of a quantity and an item type. To take a simple example, a snippet of a window production supply chain is show in Figure 1. A node with node-ID 1 needs three items of type wood to produce five items of type window frame. A node with node-ID 2 needs two items of type window frame to produce one item of type window. The actual production flow is computed by

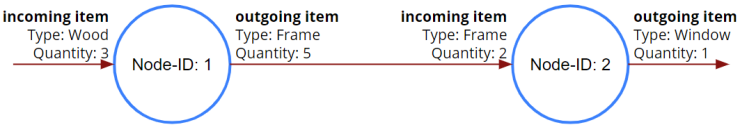

Figure 1: Snippet of a Window Production Supply Chain

a leader node and stored in a globally accessible way. During production, nodes query the flow to identify from which specific nodes they should supply items and to which nodes they should send the

resulting items. In this way, each participating node in `factoryOS` knows only about its neighbors and the leader. Hence, each stage is responsible for making manufacturing decisions based on the limited information the leader provides to the node instead of attempting to store and process the full supply-chain state in a central data store. All communication that is required for production takes place on a node-to-node basis without any leader participation.

By localizing the manufacturing decisions, `factoryOS` guarantees three out of its four system properties. First, the localization increases the robustness of the supply-chain management system and ensures fault-tolerance. Each node in the supply-chain can crash, be delayed or be paused, for e.g. maintenance, without having to halt the entire supply-chain. This is a core improvement to traditional systems where an unavailable database can halt an entire factory. We'll elaborate on this in Section 3.3.5. Second, localizing manufacturing processes reduces the dependency of having "fat servers" that need to be scaled vertically in storage and computation as the system state increases in size. `factoryOS` makes each individual node manage its incoming inventory without a clever algorithm to partition shards since the shards are created natively.

3.2 Flow

A crucial concept used by `factoryOS` is the concept of flow. The flow represents the up-to-date quantities and types of items being exchanged between the stages in the system. Extending the previous example in Figure 2, a stage can "list" itself as a node that can supply a batch of up to five frames. The downstream node does not need to consume all five frames. The flow in this case will be set to 2 out of 5. The flow does not translate to the actual requirements of the item being manufactured. It determines the batch sizes that the nodes are willing to exchange. So, the stage producing windows might request frames every 30 minutes but output a window every 2 hours. That decision is left up to the stage. After the leader has computed the flow and stored its instructions in a

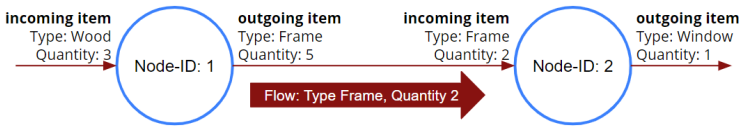

Figure 2: Flow between two nodes

globally accessible way, `factoryOS` establishes the contract between the stages. It is left up to the nodes to enforce the contract. In figure 2, the frame stage is free to deny a request from the window stage if a batch of frames is not ready. In that case, the window stage is free to go back to the Leader and ask for the flow to be adjusted or retry at a later point.

3.3 Implementation Details

Before diving into the operations that can be performed by `factoryOS`, see figure 3 that outlines all the essential components of the system. Our implementation is based on Python where we extensively use Threads and Processes to simulate a geo-distributed cluster. We briefly go over the involved components in the following sections.

3.3.1 Subscriber and Publisher

A cluster consists of different nodes, i.e. supply chain stages, that bind to a specific port. Each node is a process that has a subscriber and publisher thread, which it uses to communicate with other nodes. The publisher can broadcast messages or send them to specific nodes. The subscriber thread consumes the messages and triggers a callback in the stage or heartbeat routines.

3.3.2 Leader

One node takes the role of a leader. Only the leader has the ability to compute and manipulate the flow. We used a recursive, depth-first-search algorithm with memoization that starts at the end node and traverses through nodes until it finds a viable path to reach the start node. This allows `factoryOS` to dynamically find viable flows in case a node crashes or if requirements for a node change. The leader is elected through system file locks but can also be determined through services such as a

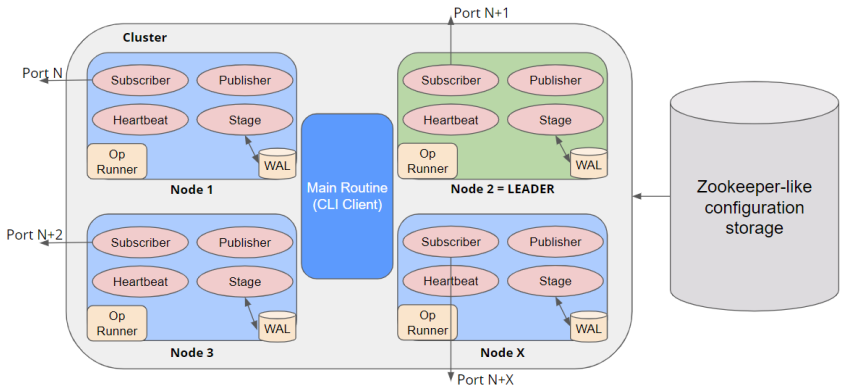

Figure 3: Implementation of `factoryOS` where each node is a Python process using the file system as configuration storage

configuration management service such as Zookeeper. In case the leader fails, we reelect the leader once nodes detect the death through heartbeats.

3.3.3 Heartbeats

Each node also contains a heartbeat thread. Heartbeats are purely neighbor-based. This means that the heartbeat thread uses the globally accessible flow graph and sends/receives heartbeats only to/from nodes it trades parts with. This drastically reduces the number of messages exchanged within the cluster. Heartbeats are used to detect crashes. For example, if a node does not receive a message from a neighbor within a certain time range, the leader gets notified by the heartbeat thread and prompted to recompute the flow. This means that only the node's neighbors are a candidates for crash detection.

3.3.4 Stage

The Stage routine is responsible for producing, requesting and sending batches of items. Stage uses different forms of signals to keep track of inventory. Every stage contains an inbound and outbound queue of items, i.e. batches items that are ready to be consumed and sent respectively. These batches are marked with a "in-queue" flag in the write-ahead log (WAL) in both the sending and receiving node once the trade has occured.

To receive a certain item, a stage sends a `RequestBatch` to a valid neighboring upstream node. The neighboring node can either send the item batch (denoted by a `BatchSent`) or respond that the item batch is not available (denoted by a `BatchUnavailable`). In case the neighbor sends the batch, the flag of this batch changes to "in-transit" in the WALs of both nodes. The downstream node also sends a `WaitingForBatch` as an acknowledgement of the trade being commited. When the batch gets delivered downstream, the receiving stage sends a `DeliveryConfirmed` message to the upstream neighbor and the batch flag changes to "delivered" in the upstream node's WAL (and "in-queue" in the downstream node's WAL. Once the batch is consumed by the downstream node, i.e. used to produce a certain item batch, the flag of the item batch changes to "consumed" in the WAL.

Using inbound queues makes `factoryOS` resistant to crashes and network delays. As long as there are items in the queue, the stage can keep producing without interruption. Knowing this, the node's parameter can be tuned to keep a certain of buffer of inventory in case of a choppy network. If neighboring nodes delay sending batches of item, nodes can bulk order items. As soon as the network gets better, the node can go back to non-bulk orders.

3.3.5 Stage Recovery

The stage thread stores all its operations in a WAL to support recovery. Every time a batch changes its status, e.g. from delivered to consumed, the stage routine persists this change in its WAL. In the case a node crashes and gets back into the flow, the stage thread will try to recover the in-queue and in-transit batches by restoring it from the WAL. In case a batch was in transit, the recovered node re-sends the a message to the downstream node and the downstream node will reply with a delivery

confirmation in case the batch was already delivered. If it was not delivered, the down stream will respond with a `WaitingForBatch` which will re-assert that the batch is in transit.

It is import to observe that the state of a batch is sequential, i.e. `in-queue` in the upstream node, `in-transit` in both nodes and then `in-queue` and `delivered` in the downstream and upstream node respectively. Therefore, any two nodes can agree on the state of the batch by picking the stage that is later in the before mentioned order.

3.3.6 Operations Runner

An operations runner thread exists in every node. Operations are different from messages exchanged between nodes. Their primary usage is to manually instruct a node to commit an action using an external client from the main thread. Operations runner is exclusively for experimentation purposes, where we require execution of multiple randomized operations e.g. Crashes, Recoveries, Updates.

3.3.7 Configuration Storage

Configuration in this case refers the potential inbound and outbound edges of a node, the flow represented as a DAG of active edges, the current leader. Since this information should be eventually consistent across all nodes, any distributed storage system that scales well for a high frequency of reads on graph like data will suffice. For example, neo4j paired with Zookeeper for leader election. Our implementation used atomic file storage with serialised Python objects given all nodes shared the file system. Our configuration management APIs are designed in a way which makes it easy to plugin popular services such as Zookeeper without any intrusive changes.

3.4 Operations

Using the described implementation of `factoryOS`, we can perform multiple scenarios within the supply chain context. `factoryOS` supports node crashes, addition of nodes, requirement changes on a node level, and live production rerouting in case a node has a maintenance operation. Given supply-chains are extremely fragile, `factoryOS` removes the challenge of interruptions during production. In short, we'll explain the node crash scenario using the example in figure 4. In case a node crashes,

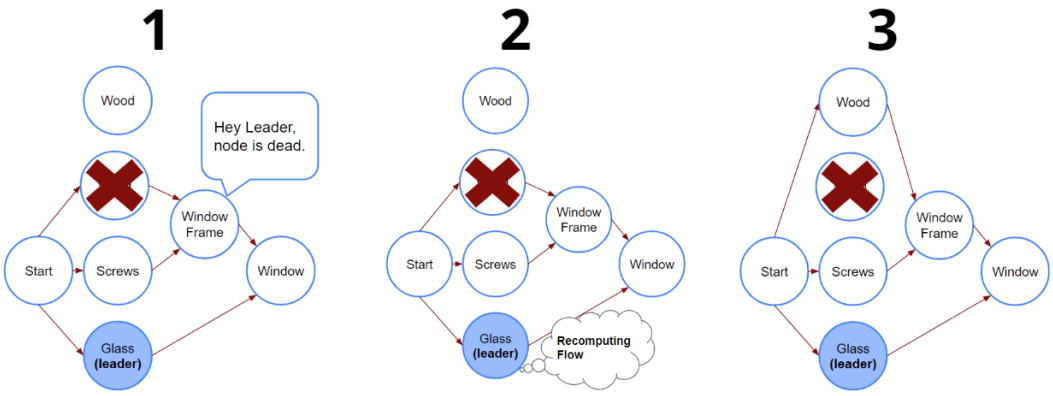

Figure 4: Implementation of `factoryOS`

one of its neighbors will detect that it doesn't receive any heartbeat messages from the crashed node. Due to lack of heartbeats, the neighbor will inform the leader that the crashed node is dead (1). The leader will recompute the flow and update it in the globally accessible file storage (2). Once that is done, the neighbors begin using the new flow. It is important to restate that all non-neighboring nodes are still able to continue with production. All neighboring nodes dependent on the crashed node are also able to continue in case they have enough batches of items produced by the crashed node in their inventory.

4 Experiments

We model supply chains with two parameters: $\mathcal{N}$ and $\mathcal{T}$, where $\mathcal{N}$ (number of nodes in active flow) is the depth of the supply chains and $\mathcal{T}$ is the number of alternative suppliers per product type.

To validate the system's fault tolerance, we randomly kill nodes and then recover them. The failure rate is defined as number of failure nodes per minutes and recover rate is defined as number of nodes to be recovered in the simulation.

We use Dual Xeon Gold 6130 CPU with total 64 cores to run `factoryOS` simulation. And the simulation is run for 10 minutes for all size of supply chains.

4.1 Performance Metrics

Metrics that determine the efficiency/scale of the system include:

1. **Successful Manufacture Cycles**: Per node average of number of manufacturing cycles in which a stage was able to successfully produce. A cycle includes consuming a item from all the inbound queues and producing one batch of the resulting item.
2. **Messages received and sent**: Per node average of the number of messages sent and received.
3. **Heartbeats** Per node average of the number of heartbeat requests, responses sent and received.

4.2 Evaluation

Refer to Figure 5 6 for the simulation of different size of `factoryOS`, it demonstrates that the `factoryOS` is scalable to all size of global supply chains. But as $\mathcal{N}$ increases, total messages sent and received grow, heartbeat messages (sent and received) grow, as a result the overall CPU usages and memory consumption increases to exhaust our hardware, slowing response to heartbeat causes neighbour nodes' false report to leader about the death of nodes, leader reconfigure the flow. As a result, the percentage of successful manufacturing cycle over total cycle decreases.

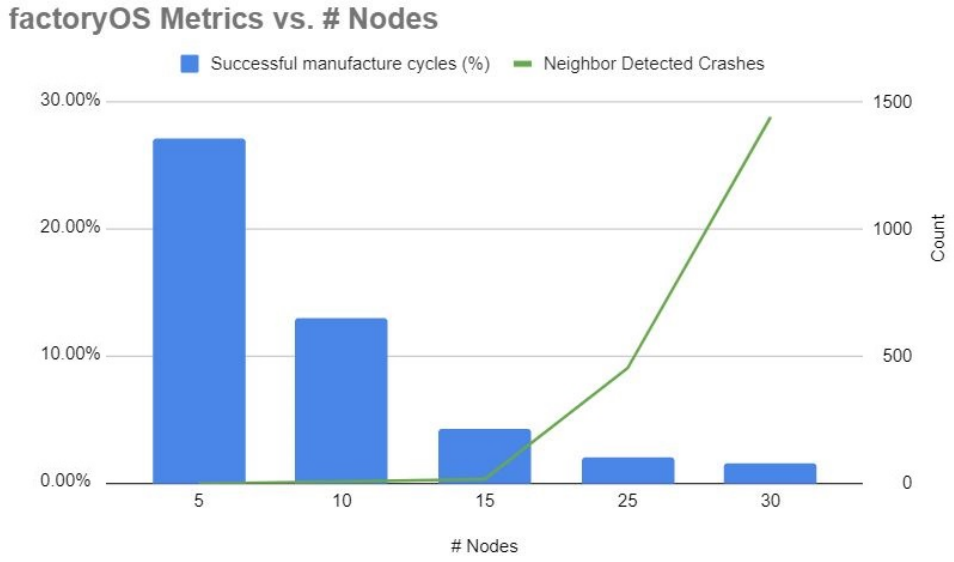

Figure 5: factoryOS scalability

5 Conclusion

Refer to Figure 7, this experiments demonstrates that `factoryOS` is able to detect the dead nodes and reconfigure the manufacturing flow using alternative suppliers. And it's robust to maintaining the successful manufacturing cycles even the rate of node crashes increases.

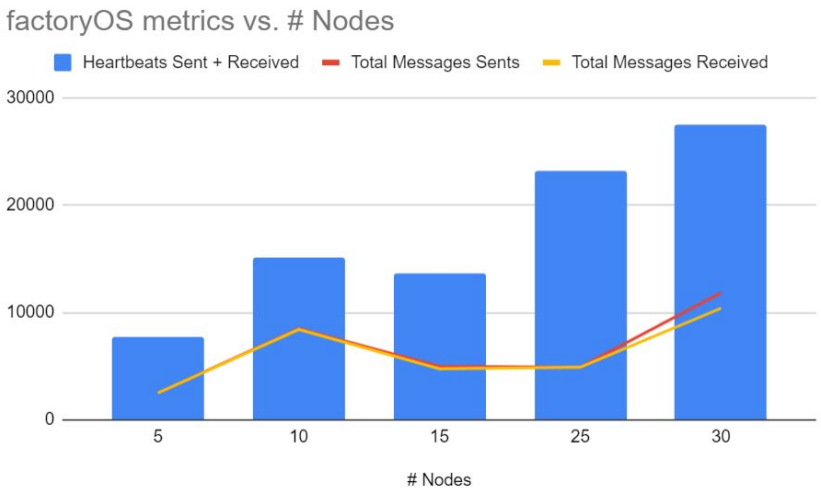

Figure 6: factoryOS scalability

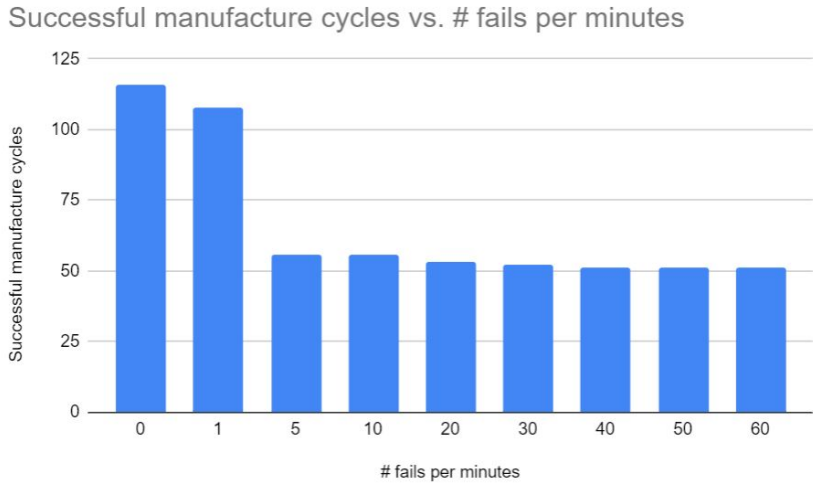

Figure 7: factoryOS fault tolerance

6 Future Work

In the future, we want to extend our solution and include ZooKeeper for coordinating a distributed configuration storage. Also, we plan to add proper leader election using Raft. Lastly, we strive to test our system across multiple machines and networks to analyze our performance in a more realistic setting.

References

[1] Saveen A Abeyratne and Radmehr Monfared. Blockchain ready manufacturing supply chain using distributed ledger. 9 2016.

[2] Alan Punter. Supply Chain Failures. 2013.

[3] Tim Conor. Still waiting for nike to do it. 2001.

[4] Marc Poulin, Benoit Montreuil, and Alain Martel. Implications of personalization offers on demand and supply network design: A case from the golf club industry. *European Journal of Operational Research*, 169(3):996 – 1009, 2006.

[5] Zhimin Gao, Lei Xu, Lin Chen, Xi Zhao, Yang Lu, and Weidong Shi. Coc: A unified distributed ledger based supply chain management system. *Journal of Computer Science and Technology*, 33(2):237–248, 2018.

[6] Christopher Reining, Omar Bousbiba, Svenja Jungen, and Michael Ten Hompel. Data mining and fault tolerance in warehousing. In Wolfgang Kersten, Thorsten Blecker, and Christian M. Ringle, editors, *Digitalization in Supply Chain Management and Logistics: Smart and Digital Solutions for an Industry 4.0 Environment. Proceedings of the Hamburg Inter*, volume 23 of *Chapters from the Proceedings of the Hamburg International Conference of Logistics (HICL)*, pages 215–232. Hamburg University of Technology (TUHH), Institute of Business Logistics and General Management, 2017.

[7] Omed Habib. 5 Tricky Sql Database Performance Challenges. 2015.

[8] N. R. Srinivasa Raghavan and N. Viswanadham. Performance analysis of supply chain networks using petri nets. In *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*, volume 1, pages 57–62 vol.1, 1999.

[9] Luis Alberto, Martínez Riascos, and Paulo Miyagi. Supervisor system for detection and treatment of failures in manufacturing systems using distributed petri nets. *IFAC Proceedings Volumes*, 34:83–88, 08 2001.

[10] J. Barata, Luis Camarinha-Matos, Raymond Boissier, Paulo Leitão, and Francisco Restivo. Integrated and distributed manufacturing, a multi-agent perspective. 11 2001.

[11] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.