Milestone 3

Param Manhas [*]
SFU 5th Year Student
paramm@sfu.ca

Harwinder Dhillon
SFU 4th Year Student
hda26@sfu.ca

Mubanga Ben Ngosa [†]
SFU 4th Year Student
mngosa@sfu.ca

Abstract

Distributed Systems Challenges to be addressed by architecture

• Consistency and replication
• Replication for fast access, scalability, avoid bottlenecks
• Require consistency management among replicas
• Fault-tolerance
• Correct and efficient operation despite link, node, and process failures

Distributed Systems Challenges (3)

• Distributed systems security
• Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
• API for communications, services
• Ease of use
• Scalability and modularity of algorithms, data, services
• Decentralized logic, caching

Distributed Systems Challenges (4)

• Transparency: hiding implementation policies from the user
• Access: hide differences in data representation across systems, and provide uniform operations to access resources
• Location: locations of resources are transparent
• Migration/Relocation
• relocate resources without renaming
• relocate resources as they are being accessed
• Replication: hide replication from the users
• Concurrency: mask the use of shared resources
• Failure: reliable and fault-tolerant operation

[*] with optional author note
[†] with optional author note

CCS Concepts **• Software and its engineering → General programming languages**; **• Theory of computation** → *Program analysis*

Keywords

1. Design Details

Data Storage and Access:

Sub-Task: Implement a distributed data storage system that supports fast and scalable access across the network.

Sub-Task: Develop a replication strategy (such as data partitioning or others) to ensure fast access, scalability, and fault tolerance. Select where replicas should be placed within the cluster. Options include random placement, colocating replicas on the same rack for fault tolerance, or using a specific policy like quorum-based placement.

Solution: Implement a distributed data storage system based on technologies like Kubernetes. Utilize Kubernetes distributed storage solutions like MinIO, Ceph, for managing data storage and access.

Evaluation: Measure data storage and retrieval latency, throughput, and scalability under different loads and network conditions. Conduct tests with various data sizes and access patterns to assess the system's performance. Measure the system's response time under different levels of replication and assess its ability to maintain data consistency.

Consistency and Replication:

Sub-Task: Develop a replication strategy to ensure fast access, scalability, and fault tolerance with Kubernetes.

Solution: Kubernetes StatefulSets to manage application replicas, allowing for consistent naming and identity. Implement application-level replication for data consistency. Use Kubernetes ConfigMaps and Secrets for configuration management, making it easy to maintain replica configurations. The downside is the data stored in a ConfigMap cannot exceed 1 MiB max

Evaluation: Measure the system's response time under different levels of replication. Test the system's ability to maintain data consistency and availability in a Kubernetes-managed environment.

Fault-Tolerance:

Sub-Task: Implement mechanisms for detecting and recovering from link, node, and process failures in a Kubernetes context. Implement mechanisms for detecting and recovering from link, node, and process failures using Kubernetes features such as Readiness Probes, Liveness Probes, Pod Disruption Budgets, and Horizontal Pod Autoscaling.

Solution: Utilize Kubernetes features like readiness and liveness probes to detect node and process failures. Implement PodDisruptionBudgets to control disruptions. Employ horizontal pod autoscaling to handle increased load and node failures automatically.

Evaluation: Simulate various failure scenarios, including Kubernetes node or pod failures, to validate the system's ability to operate correctly and efficiently during these events. Example Scenario: A node in your Kubernetes cluster becomes unresponsive, and you want to ensure your application continues to function without disruptions.

Solution: You can use the Kubernetes Liveness Probe to detect node failures. If a node goes down, the pods running on that node will fail their liveness probes, leading to their automatic restart on other healthy nodes.

Secure Channels and Access Control:

Solution: Implement robust security measures to protect sensitive graph data. Utilize encryption for data in transit and at rest. Use authentication and authorization mechanisms to control access. For example, Kubernetes supports RBAC (Role-Based Access Control) to manage access permissions. Secure communication channels between distributed components using TLS/SSL certificates. Use mutual authentication for secure API access control. Kubernetes provides built-in support for securing communication between pods and services. Implement a centralized key management system for generating, distributing, and rotating encryption keys. Possibly use Kubernetes Secrets to securely store and manage keys.

Evaluation: Assess encryption success with unit tests. Verify that communication channels are appropriately secured and that access control mechanisms are effectively managing user and service permissions.

API for Communications and Services:

Solution: Develop a well-documented API that provides a clear interface for communication and services. Consider implementing GraphQL APIs for data retrieval and analysis.

Evaluation: Conduct usability testing to ensure that the API is user-friendly and satisfies the needs of the application. Monitor API performance and scalability.

Scalability and Modularity of Algorithms, Data, Services:

Solution: Design algorithms, data structures, and services to be modular and scalable. Utilize Kubernetes Horizontal Pod Autoscaling to automatically adapt to varying workloads. Evaluation: Conduct load testing and scalability assessments to ensure that the system can efficiently handle increased data and analysis demands.

Decentralized Logic and Caching:

Solution: Implement decentralized logic for distributed components to independently manage graph mining tasks. Use caching mechanisms like Redis for intermediate results. Evaluation: Evaluate the performance and resource utilization of decentralized logic. Measure the impact of caching on data retrieval and analysis speed.

Concurrent Operations and Conflict Management :

With multiple operations, the system should handle simultaneous tasks on shared resources without any data conflicts. The Concurrency Manager is developed to facilitate this, avoiding potential conflicts during simultaneous resource access. Its reliability is validated by introducing concurrent operations and ensuring that they don't result in conflicts. Additionally, simulating potential deadlock scenarios can attest to the system's preventative measures.

Resource Mobility and Optimization:

A seamless user experience involves the ability to relocate resources without interruptions or name changes. The Migration Controller is in charge of overseeing and executing migrations based on various criteria. By simulating multiple migration scenarios, its success rates can be determined, ensuring resources aren't renamed and downtime is minimized. Additionally, the Dynamic Relocator functions to allow the live relocation of resources during active access. Continuously accessing a resource during its relocation serves as a means to evaluate any potential disruptions or inconsistencies.

Data Integrity and Accessibility:

Definition: The essence of this challenge lies in offering consistent and unified operations to users irrespective of the backend data representation. A Data Access Layer (DAL) provides standardized operations for data access. By making a variety of CRUD requests and benchmarking operation latencies, we can assess the accuracy and speed of the DAL. Additionally, the Replication Manager plays a pivotal role in transparently handling data replication and ensuring data consistency across all replicas. By comparing data after replication events and measuring replication time, its effectiveness is scrutinized.

Thread Communication

Sub-Task: Implement communication between nodes.

Solution: Use the gRPC framework to build asynchronous communication between processes and nodes. Use communication methods as unicast messaging and broadcasting for transferring data and shutdowns.

Sub-Task: Distribute work across all nodes and processes in a way that does not cause load imbalance.

Solution: Issue the work (vertices to be explored) in a concurrent queue or an atomic node pointer that points/pops off the. Processes that are free to work, will just grab the work in chunk sizes that will gradually decrease as computation progresses.

Evaluation: We will set up tests that will monitor thread progress and inactivity among various graphs and patterns.

Global Snapshot

Sub-Task: Global Snapshot For Recovery

Use chandy lamport snapshot algorithm for recording global snapshot in the system. We will capture the local state of all processes including (vertices explored, unexplored vertices, discarded vertices, paths found)

Evaluation: We will set up tests that will have a set of possible results in states and see if recovery is possible.

A. Appendix Title

Acknowledgments

References

K. Jamshidi, R. Mahadasa. and K. Vora. ...Peregrine: A Pattern-Aware Graph Mining System...