Pioneers

Anastasiia Skrypnykova	22207108	Tong Wu	22205061
Jiajing Li	22200375	Saoirse Duignan	19326976
Xiuping Xue	22200549		

Synopsis

Distributed Health Monitoring and Alert System (DHMAS) is a system designated to monitor patients' health in real-time and issue alerts based on collected data. It integrates with various wearable devices and health sensors to track the status of the patients.

Our application can prove its value in the medical domain, where numerous patients need to be monitored constantly, and missing critical conditions will have a significant impact.

The main idea of DHMAS is to provide medical practitioners with a reliable tool to track and store the conditions of a vast amount of patients simultaneously. Our app sends notifications about critical status and allows to view and filter all the disturbing patients on the dashboard.

Technology Stack

- ***MQTT (Message Queuing Telemetry Transport)***

  A protocol, that is commonly used in the Internet of Things (IoT) applications, where a large number of devices and sensors send data to the server for further data usage. In our app, four medical devices were modelled (holter, respirometer, oximeter, and thermometer). They are designated to generate random values every 15 seconds for ten patients, where each value corresponds to the producer device.

- ***Mosquitto***

  An open-source broker that implements the MQTT protocol [1]. It provided us with the necessary infrastructure to facilitate the communication between the modules. It is a central server to which IoT “devices” (holter, respirometer, oximeter, and thermometer) connect and publish messages. At the same time, it also allows the PDCS module to subscribe and receive those messages.

- ***Apache Kafka***

  In the data transmission between PDCS and HAS, as well as between HAS and ANS, we have utilised Kafka [2] as our transport queue. This choice was motivated by Kafka's outstanding capabilities in handling large data streams, high throughput, and reliability.

  Our application primarily focuses on real-time data processing, which is also a feature supported by Kafka. This enables our system to analyse data in real-time and send alert messages promptly and effectively.

- ***MyBatis***

The interaction between the HAS service and the database utilises MyBatis, enabling a variety of database operations. MyBatis was chosen because it can be easily integrated into existing Java applications and allows SQL statements to be directly defined via XML, without the need for complex configuration. This makes it well-suited to the design requirements of the HAS service, facilitating the modification of data received from PDCS and its subsequent insertion into the database.

- ***RESTful API (REpresentational State Transfer Application Program Interface)***

  RESTful APIs are well-suited for CRUD (Create, Read, Update, Delete) operations in a Spring Boot application using JPA (Java Persistence API) to interact with a database. RESTful principles align closely with the standard HTTP methods, making it easy to map CRUD operations to HTTP methods. RESTful APIs use URIs to uniquely identify resources. In a Spring Boot application, these URIs can be mapped to endpoints that correspond to entities in the JPA-managed database.

  This URI structure is intuitive and aligns with the structure of the underlying data. Spring Boot provides excellent integration with Spring Data JPA, which simplifies the implementation of CRUD operations. Spring Data JPA provides high-level abstractions and annotations that allow us to create repositories and perform database operations without writing boilerplate code.

- ***Java with Spring Boot***

  Primary language and framework for backend development due to its robust ecosystem and support for building scalable microservices. SpringBoot simplifies the bootstrapping and development of new Spring applications.

- ***React***

  React framework is used in the PMD part. It is a front-end JavaScript library that allows us to build user interfaces from reusable UI components. React uses server-side rendering to provide a flexible and performance-based solution. It allows developers to create seamless UX and complex UI. Since React follows the “Learn Once, Write Anywhere” principle, it was the preferred choice to build a fast and scalable application for us.

- ***PostgreSQL***

  Chosen for its performance in read-write operations and support for complex data types.

- ***Docker***

  To automate the deployment of our application inside lightweight, and portable containers, Docker was chosen. Containers package an application and its dependencies, ensuring consistency across different environments. With Docker, we deployed our entire app with all its dependencies, which significantly reduced the complexities of the deployment.

- ***Kubernetes***

  Building on top of Docker, we also utilised Kubernetes to manage containerized applications by packaging our different services into containers and deploying them to a cluster. Kubernetes provides built-in service discovery and load balancing capabilities, enabling different components to discover and communicate with each

other, and automatically distributing traffic to available replicas, thus facilitating load balancing. [3]

Design Considerations

In designing the DHMAS, we considered several critical factors to ensure its effectiveness and reliability. We use the ***CAP theorem***, and our system focuses on Consistency and Partition Tolerance (**CP**).

Given the nature of healthcare and the need for real-time monitoring and alerting, our system prioritises data consistency. This is achieved by sharing one centralised database and having a single datasource (PDCS).

The system is designed to tolerate network partitions by using Kafka. Kafka doesn't need the clients and servers to be online at the sametime. Therefore, in scenarios where network failures divide the system, DHMAS can continue to operate effectively in isolated partitions, ensuring continuous patient monitoring and alerting,

System Overview

At its core, the DHMAS system has five components:

- **Patient Data Collection Service (PDCS)**
  Acts as a Kafka producer to stream simulated health data in real-time, ensuring that the data is consistently formatted and valid.

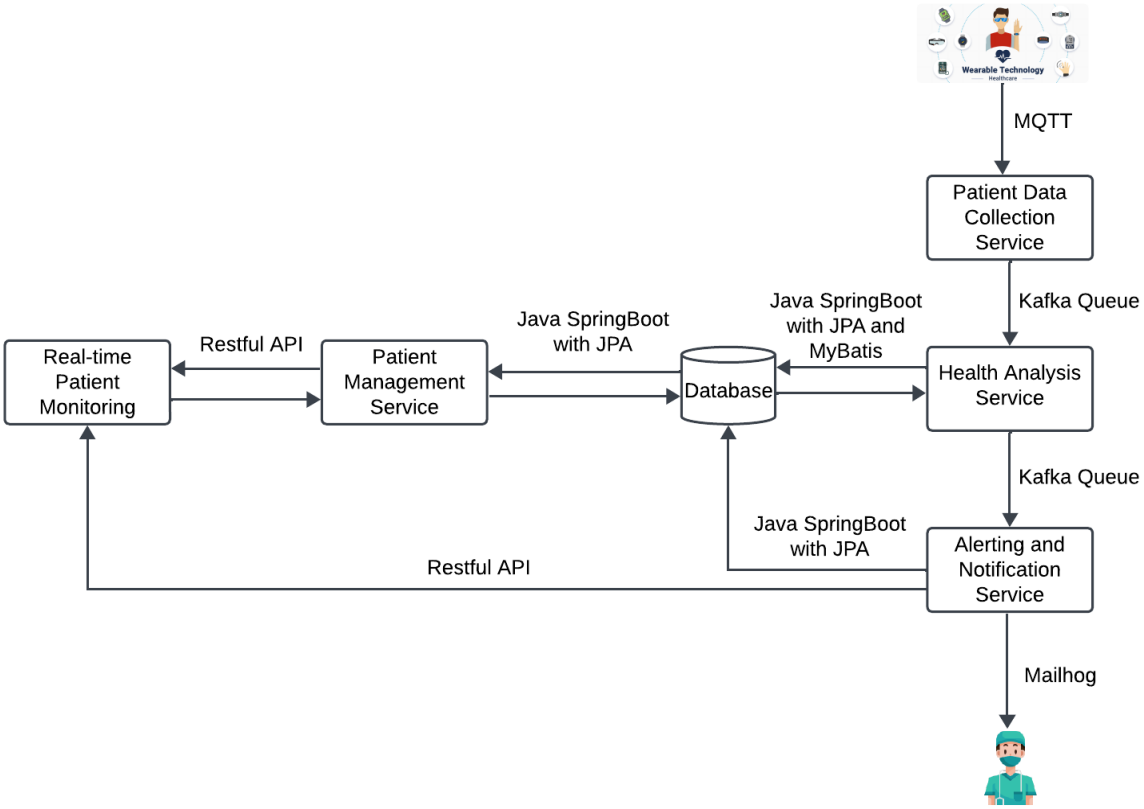
- **Health Analysis Service (HAS)**
  HAS initially serves as a consumer of the PDCS message queue, assessing the real-time status of patients and marking messages as normal or abnormal according to a pre-set rule, and then writing them into the database. For all abnormal messages, HAS will retrieve the patient's general information from the database using their ID. This information is combined with the real-time data from the abnormal message to construct an alert message. Then, HAS acts as a Producer for another Kafka queue and sends this alert message to ANS.

- **Alerting and Notification Service (ANS)**
  Receives data from the HAS module via Kafka and sends email notifications to those patients with the 'abnormal' status. Stores the received data and messaging information into a database for future reference and exposes an API for sending emails.

- **Patient Management Service (PMS)**
  Manages patient profiles utilizing Get and Post Requests with CRUD operations, along with authenticating users this provides security.

- **Real-time Patient Monitoring Dashboard (PMD)**
  Show the summary of the whole data and patient profile, visualise data via charts and add new patients into the database through the Dashboard.

System Architecture Diagram



The entry point of the program is the Patient Data Collection Service (PDCS). It collects information from the medical devices that each patient wears constantly. Those medical devices include a Holter, which sends information about heart rates, a Respirometer, which is responsible for collecting respiratory rates, an Oximeter, which measures and sends data about oxygen saturation of the blood (SpO2), and a Thermometer, which transmits data regarding patient's temperature. The combined information about each patient is then transmitted via MQTT protocol to the broker, Mosquitto, which in turn connects to the PDCS module. To establish communication between PDCS and the next module, Health Analysis Service (HAS), Kafka Queue was used.

HAS is responsible for analysing the dynamic data and finding potential and obvious problems within the data. Besides, HAS will mark and insert the received data into the central database using MyBatis. If any risk factors are found during the analysis process, HAS will combine the risk information and general information together and signal to the Alerting and Notification Service (ANS) using another Kafka Queue.

When ANS receives the alerts, it adds them to the database first. Then ANS sends emails to doctors to inform the details about patients who are at risk. Medical practitioners can use the alerts and data they receive to determine if a patient needs urgent help.

At the same time, these risk alerts are also displayed on the dashboard within Real-time Patient Monitoring (PMD). PMD can obtain basic information about patients, such as age, gender, and type of disease, from Patient Management Service (PMS), which is responsible for providing API for the Create, Read, Update, and Delete operations on the database that is shared among all other modules. In this case, healthcare data administrators and doctors can add new patients via PMD easily. PMD can also send customised messages to given

patients using API from ANS, which is convenient for further diagnosis and communication with the patient.

Scalability and Fault Tolerance

Microservices

DHMAS conforms to the definition of a microservice, achieving service independence, technological diversity, fault tolerance, and resilient design. In this microservice architecture, each service is independent and can be deployed and modified separately. This allows us to scale up or scale down resources independently for each service according to its needs, without impacting other parts of the entire application. Additionally, microservices allow the use of different technologies for different services, which further supports scalability. In addition, if a service fails, it can be independently restarted or fixed while other modules continue to operate, thus enhancing the overall fault tolerance of the system.

The use of Kafka and Zookeeper

As mentioned above, in our application, new data for each patient is generated every 15 seconds, leading to a substantial volume of data. In real-life scenarios, our application might serve an even larger number of patients, facing more intense demands for message transmission. Kafka, which supports high-speed data streams and high throughput, makes it an apt choice for our transport queue to meet these demands.

High data reliability and persistence are also ensured by using Kafka, which replicates data across multiple nodes in the cluster. This feature safeguards against data loss even in the event of single-point failures, thereby enhancing the stability of our message queue and improving the fault tolerance of our application.

Kafka was used to distribute the load across multiple servers. It is particularly useful in scenarios, where different patients' information needs to be available across various servers for separate processing.

At the same time, our Kafka queue relies on Zookeeper to maintain its cluster metadata and state information. [4] This means that our system also leverages the advantages of Zookeeper to enhance scalability and fault tolerance. For example, a Zookeeper can use a leader election algorithm to deal with fault, thus ensuring the continuity of the service. Meanwhile, Zookeeper holds the key configuration information of the system, so we can easily add new nodes when expanding the system while maintaining the consistency of configuration information, which increases the overall stability of the system.

Duplicate data storage

As a microservices system, duplicate storage can add an extra layer of fault tolerance. Specifically, the ANS and HAS both stores messages about the ‘abnormal’ patients, where the ANS stores it in the ‘alert_message’ table and the HAS stores it in the ‘status_localis’ table. This redundancy serves as both a reference and a storage mechanism, ensuring that critical information is not lost in case of a single service failure. For instance, if one service that processes alert messages encounters an issue, the duplicate messages stored in the alert_message table can be utilised by another service to ensure continuous operation.

The use of Docker

To facilitate easy scaling of the application, we divided it into smaller, independently deployable modules. For this, we used Docker containers, which provide isolation between

all of the modules. This not only helped to avoid conflicts between different modules but also allowed for the scaling of each individual module in the container.

Moreover, containers can be rapidly started, stopped, and replicated. This quick deployment capability allows for a quick response to changes in demand, enhancing the system's scalability. Furthermore, when used in conjunction with Kubernetes, Docker can automatically balance the load among various containers. This ensures optimal utilisation of resources and maintains performance during periods of high load. Docker also enables more efficient use of system resources, allowing for more applications to run on the same hardware, thereby contributing to improved scalability.

The use of Kubernetes

We also provide a Kubernetes version of deployment. By using Kubernetes, we can automatically scale the number of container replicas based on the service's needs, dynamically adjusting the scale of applications to meet different load conditions, thereby enhancing scalability. Secondly, Kubernetes uses ReplicaSets to ensure the automatic replacement of containers in case of failure, maintaining the desired number of replicas running. This contributes to improved fault tolerance. Additionally, Kubernetes allows us to configure resource limits and requests for different workloads, achieving isolation and allocating sufficient resources for critical tasks. This helps improve application fault tolerance and avoid resource contention. Most importantly, Kubernetes clusters are typically composed of multiple nodes, some of which can fail without affecting the overall cluster's availability. This helps enhance our application's fault tolerance and stability.

Contributions

Anastasiia Skrypnykova

Implemented the PDCS module.

Created a data generator to simulate the real-time working medical devices, one for each recorded property:

- Holter: generates random heart_rate every 15 seconds for 10 patients.
- Oximeter: generates random SpO2 every 15 seconds for 10 patients.
- Respirometer: generates random respiratory_rate every 15 seconds for 10 patients.
- Thermometer: generates random temperature every 15 seconds for 10 patients.

Employed the use of MQTT clients and brokers.

Designed a Kafka service to forward the received messages from the MQTT broker to the HAS module.

Jiajing Li

Implemented the HAS module.

HAS stands for Health Analysis Service.

- Serves as a Consumer of one Kafka queue and receives data from PDCS.
- Ensures a smooth connection between PDCS and HAS.
- Evaluates the received data, adds attributes, and writes to the database using MyBatis.

- Combine information from two tables to generate an alert message.
- Acts as a Producer for another Kafka queue and sends the newly assembled alert message to the ANS service.

Xiuping Xue

Implemented the ANS module.

The Alerting and Notification (ANS) Service:

- Receives data from the HAS module using Kafka.
- Sends emails to those altered patients.
- Uses MailHog to monitor sent emails.
- Stores received data and sends message information into the database.
- Exposes an API for sending altering emails.

Saoirse Duignan

Implemented the PMS module.

PMS stands for Patient Management Systems (PMS). This part modifies and changes the tables.

- Handled CRUD Operations: The PMS module manages all Create, Read, Update, and Delete operations for patient profiles.
- Intermediary Role: It acts as an intermediary between the tables and the information.
- Utilised Get and Post Requests: The module implements these operations through Get and Post requests, using Postman for testing purposes.
- Implemented Security Features: To ensure the security of the system, especially vital in healthcare, implemented authentication measures in Spring Boot.
- Created Login Credentials: Developed a system requiring a username and password for editing or changing information in the database, enhancing the system's security.

Tong Wu

Implemented the PMD module.

PMD stands for Patient Management Dashboard. It is responsible for visualising patients' information.

- Summarise the total number of different categories.
- Display all patients' personal information as a table.
- Add filters to show what users want to focus on.
- Add a component to insert new patients into the database.
- Count and Visualise patients' information as charts.

Reflections

Challenges and Solutions

<u>Tech Section</u>

One of the main challenges we encountered was how to uniform data formats, including communication between Kafka queues, as well as database operations across different services. Communication between services requires consistency in the number of attributes and data formats between both parties, but during the development process, we were working independently, which led to various difficulties when merging the first version of the code. For example, Gson used in PDCS and Json used in HAS have different approaches and specifications when handling datetime data, requiring the addition of adapters in the code for proper conversion to facilitate communication. Additionally, inconsistent capitalization of attributes' names has also caused some issues. Therefore, we documented standardised formats in our group Google Docs and made adjustments to ensure smooth connections between different services.

Another challenge was how to Dockerize various services. In this project, we attempted some new technologies, which led to many difficulties in the containerization process. For example, when packaging MyBatis projects, Maven considers src/main/java only as a source code path and ignores the mapper.xml files placed there during packaging. Therefore, additional modifications to the pom.xml file were required to ensure that these resource files were included.

For database operations, there was a challenge to handle the foreign keys in the tables. The status_localis table has a foreign key to general_info, which means one had to populate and ensure the general_info table was full before editing or changing the status_localis table. This was solved once the tables were correctly established after testing using the Postman tool.

While dealing with the PMD module, a couple of issues were also dealt with. The first one is related to the security of our application. Although we already had all the CRUD Restful APIs from the PMS part, we did not reach the desired level of data protection. To solve this issue, an authentication token was inserted into the message header of a GET request.

```
const response = await fetch (apiUrl, {
method: 'GET',
headers: {
'Authorization': `Basic ${basicAuthToken}`,
'Content-Type': 'application/json',
},
});
```

Another problem was discovered when we were trying to fetch the data. CORS, which stands for Cross-Origin Resource Sharing, is a security feature implemented by web browsers to control requests made across different origins (domains). This policy prevents our web page from making requests to a different domain than the one that served the web page, to enhance security. However, we were able to identify and tackle this error. The solution included the modification of the security settings within our SpringBoot parts.

```
protected void configure (HttpSecurity http) throws Exception {
	http
		.csrf().disable() // Disable CSRF
```

```
//                    .authorizeRequests()
//                    .antMatchers("/api/**")
//                    .authenticated() // Require authentication for API endpoints
//                    .and()
                .httpBasic(); // Use HTTP Basic Authentification
}
```

The following challenge was related to the database design. The confusion was caused by unique identifiers for patients. Since all of the existing patients already have unique IDs, it is complicated to assign IDs to the new patients added via the dashboard. To tackle this, the data type of the patient's ID was converted from INT to SERIAL when initialising the primary table. We also used the 'setval' function within our SQL statement to set the next value of a sequence in PostgreSQL. Thus, now IDs can increase automatically via this setting.

```
CREATE TABLE general_info (
                            patient_id SERIAL PRIMARY KEY,
                            age INT,
                            gender VARCHAR(10) CHECK (gender IN ('Male', 'Female')),
                            diagnosis VARCHAR(255) CHECK (diagnosis IN ('Heart
Disease', 'Cancer', 'Diabetes', 'Influenza', 'Hypertension')),
                            medication VARCHAR(255) CHECK (medication IN ('Aspirin',
'Antibiotics', 'Chemotherapy', 'Statins', 'Insulin')),
                            insurance_type VARCHAR(50) CHECK (insurance_type IN
('Private', 'Uninsured', 'Medicaid', 'Medicare')),
                            treatment_duration INT,
                            email VARCHAR(255) NOT NULL
);

PERFORM setval('general_info_patient_id_seq', COALESCE((SELECT MAX(patient_id)+1 FROM
general_info), 1), false);
```

Last but not least, the biggest challenge of this project was applying distributed technologies that were not covered in our coursework, such as Kafka queues, MQTT, and Kubernetes, not to mention React for the front-end. Initially, these technologies were just vague concepts for us, but within a limited timeframe, we managed to understand their principles through studying relevant videos and documentation. We then interconnected these technologies and, in conjunction with what we learned in class about Docker, SpringBoot, Restful APIs, and others, successfully applied them to our project.

Project Management Section

One of the challenges our team faced was a lack of communication. This ended up postponing the beginning of the project and changing the initial plan of the project. This issue was solved by using the additional time of the extended deadline and organising several virtual meetings.

If we could do it all over again, we would have stipulated the technology used for connecting services and the data formats right from the beginning of the project. This would have helped us avoid a significant amount of unnecessary adjustment work later on.

In addition, we would have conducted more research at the beginning of the project and chosen the optimal way for services to communicate before starting to write code, it would help us save a lot of time as we once managed to use WebSocket and gRPC for

communication, but we decided to replace them during the optimization process. This way, we might have the opportunity to add more services.

Also, setting up the hard deadlines for the minimum viable product at the beginning of the project would have helped us to stay on the right track with everyone contributing equally each week.

What We Learnt about the Technologies

MQTT

Getting to know the protocols of the Internet of Things was a valuable experience. Simple and lightweight protocols, such as MQTT, are perfect for data exchange with low-powered medical devices. Its Publish-Subscribe pattern allows each client to send messages that are associated with a specific “topic” to the broker. Other clients can subscribe to the topics to receive the relevant messages. There is also an option to subscribe to all the available topics using the “#” wildcard.

As for the second most important entity in the MQTT architecture, it was realised with the use of Mosquitto. MQTT broker is responsible for facilitating transactions between MQTT clients. Open-source Mosquitto broker can be easily installed and accessed via the “eclipse-mosquitto” image in Docker. However, there are a couple of drawbacks that such a broker possesses. When deploying the Mosquitto broker in a docker container, we need to be cautious of conflicting User IDs (UIDs) and Group IDs (GIDs). Mosquitto runs under the Mosquitto user with UID 1883. If the host machine has a user with the same UID, it may lead to unintended file ownership, which can compromise data security.

Kafka vs. WebSockets

During the development process, we initially set up WebSocket communication between PDCS and HAS modules. However, we learned that WebSocket is more suitable for real-time communication and facilitates user interaction.

In the design of our application, the communication between PDCS and HAS modules is unidirectional. This means that after receiving messages from PDCS, HAS does not need to interact with PDCS again. Therefore, the characteristics of WebSocket are not needed in this case.

Moreover, WebSocket communication is transient and requires **both parties** in the message queue to be online simultaneously, and it does **not involve data storage**. For this reason, we chose to use Kafka instead.

Kafka offers **data persistence**, allowing Consumers to receive messages sent by Producers even after the Producers have gone offline. In our case, it means that all the modules can still get the information, even if the medical devices are turned off. Additionally, even if the Consumer is not present, the Producer can independently send messages, thus realising a distributed design. Therefore, Kafka aligns more with the requirements of our project.

However, Kafka also has some limitations, as it offers eventual consistency rather than strong consistency. This means that in certain situations, messages might be lost or duplicated. Moreover, as we previously mentioned in the tech challenge section, Kafka does not provide built-in mechanisms for data transformation and data format validation. This may require additional development efforts to ensure the accuracy and consistency of the data, which indeed cost us a significant amount of effort.

gRPC vs. RESTful API

In our initial plan, the ANS module and the front end were supposed to communicate via gRPC. gRPC uses HTTP/2, which allows for **lower latency** and **better data compression** compared to HTTP/1.1. It is useful in microservices that need high speeds. However, as the development progressed, we found out that gRPC's configuration was relatively complex, for example, in Java we have to use pom to generate codes and write the implementation codes. The process may meet problems on different platforms. Our front end didn't actually have high-performance requirements. What we needed were simple and easy-to-use interfaces. Additionally, RESTful APIs have the advantage of being easier to test.

Although RESTful API is known for being unsuitable for complex transactions, which leads to redundancy when transmitting large amounts of data. However, they are still very **suitable for simple CRUD** operations in our system. Therefore, using REST APIs turned out to be more appropriate.

In summary, we realised that when choosing communication methods between components in a distributed system, it is important to consider practical needs such as performance requirements, real-time capabilities, scalability, and compatibility with existing infrastructure. We should select the appropriate method based on different use cases and requirements.

Kubernetes

Furthermore, we also learned about Kubernetes and mastered its use for automated deployment, scaling, and management of containerized applications. Although our project was just a microservices application, through this process, we experienced Kubernetes' ability to detect and replace faulty containers and understood how it ensures containers can communicate with each other. Considering that Kubernetes has become the standard tool for cloud-native application deployment, this is a valuable experience for our future distributed system development.

References

[1] Light, R. A. (2017). Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*, *2*(13), 265.

[2] Kreps, J., Narkhede, N., & Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (Vol. 11, No. 2011, pp. 1-7).

[3] Kubernetes Documentation. (n.d.). Retrieved from https://kubernetes.io/docs

[4] Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.