Research and activities 2023-2024

Cristi Ene

Table of Contents

This document describes the research and design of a real time distributed system used to collect data from sensors and serve concurrent requests from multiple clients (mobile, web, embedded).

Distributed system general overview

Overview of modules

This document provides an overview and detailed documentation of the distributed system comprising several modules designed for efficient data acquisition, processing, and dissemination.

High level diagram and components:

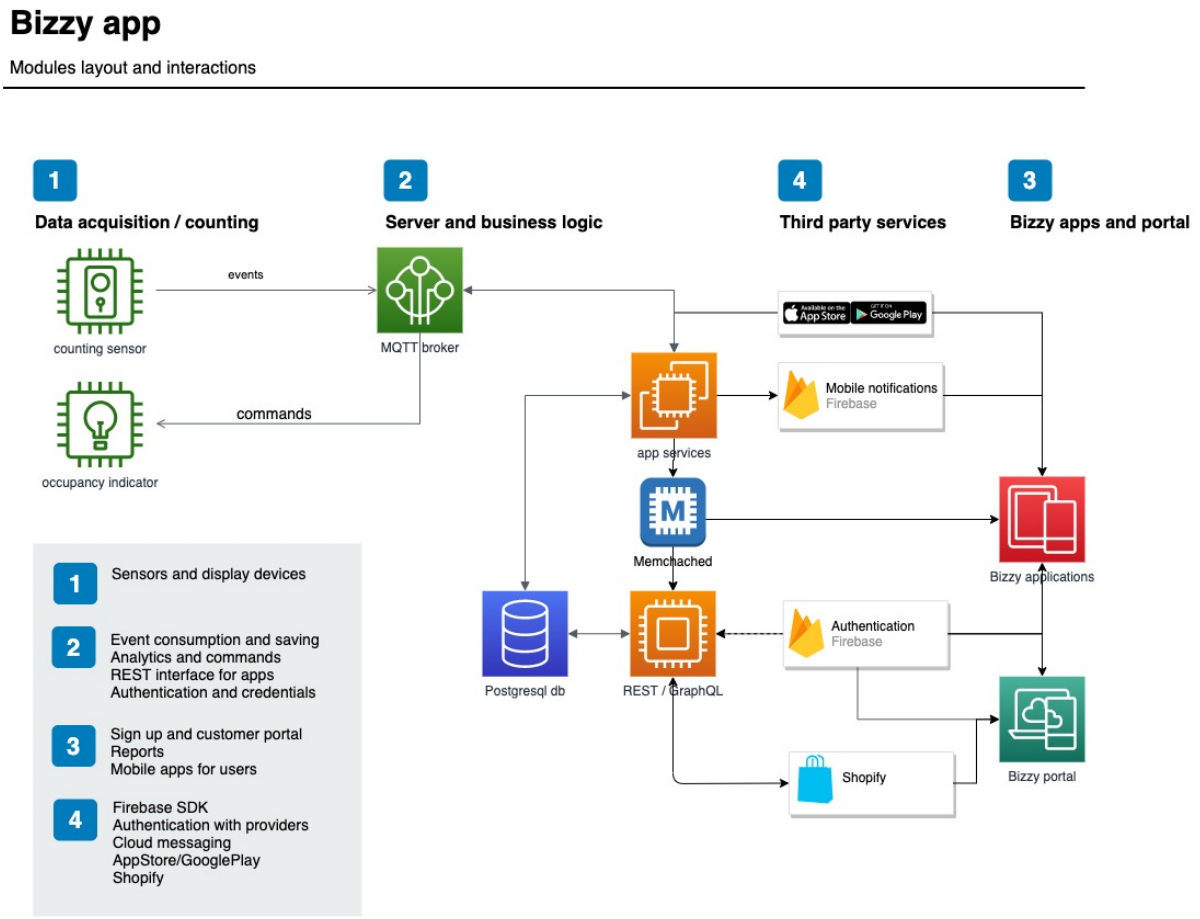


Figure 1. diagram

Modules

- Data Acquisition Sensors
- MQTT Broker
- App Services
- Client Apps

Data Acquisition Sensors

Description

The Data Acquisition Sensors module is responsible for collecting data from various physical locations. These sensors are strategically placed to capture relevant data points as per the system requirements.

Functionality

- Collects data from designated locations.
- Transmits data securely to the MQTT Broker.

Implementation Details

- Sensor types and specifications.
- Communication protocols.
- Data transmission frequency and format.

MQTT Broker

Description

The MQTT Broker serves as the central hub for receiving data from the Data Acquisition Sensors and distributing it to the appropriate App Services.

Functionality

- Receives data from sensors via MQTT protocol.
- Publishes data to subscribed App Services.

Implementation Details

- MQTT Broker configuration.
- Quality of Service (QoS) levels.
- Security measures (authentication, encryption).

MQTT broker is a RabbitMQ instance.

It receives data from sensor and puts it in a queue, it will be consumed from the queue by a bizzi component.

App Services

Description

App Services are responsible for processing data received from the MQTT Broker and performing necessary operations such as storage, analysis, and serving data to client applications.

Functionality

- Reads data from MQTT Broker.
- Saves data to the database.
- Provides data processing and analysis capabilities.
- Serves data to client applications.

Implementation Details

- Database schema.
- Data processing algorithms.
- API endpoints for client communication.

Firebase

Google Firebase to use for: - authentication - SDKs exist for implementation of authentication based on: Google, Facebook - will cut down the development time (sign in, reset passwords, authentication) - the app will not store any user data, just an `user_identifier`, all the details are on Firebase - generous limits - we could implement this ourselves when/if the user base exceeds their limits - cloud messaging (notifications) - getting notifications even if the app is closed and not in memory (but user is authenticated) - crashlitics (for error logs and reporting)

Memcached

Because the Bizzy app have features accessible even without an account, it needs to be able to allow a high number of requests. So it needs a *replica* of the data that can be easily queried without hitting the database. - all of the (free) user user calls will be answered using the Memcached service - data can be a couple of minutes older but will eventually update (hive count) - the Postgres database will be the *source of truth*

Client Apps

Description

Client Apps interact directly with end-users and provide access to the system's functionalities. These apps are designed to be versatile, catering to various platforms and device types.

Functionality

- Integrate with notification services.
- Support payments.
- Present data to users in a user-friendly manner.

Implementation Details

- Supported platforms (Web, Mobile, Embedded).
- Notification service integration.
- Payment gateway integration.

Notifications

- iOS subscriptions are not available across platforms (only (up to 5) devices under the same Apple account)
- server side notifications when a subscription is bought

Conclusion

This documentation provides a comprehensive overview of the distributed system and its constituent modules. Each module plays a crucial role in ensuring the system's functionality and efficiency. Detailed implementation details aid in understanding the system's architecture and facilitate seamless integration and maintenance.

Sensor

Message format

Bit	+0..7	+8..15
0	diff IN	
16	diff OUT	
32	total IN	
48	total OUT	
64	years	months
80	days	hours
96	minutes	seconds
112	report type	0xFF

Figure 2. sensor message

Event	Explanation
`diff IN`	Number of ENTRY events since last sent attempt
`diff OUT`	Number of EXIT events since last sent attempt
`total IN`	Total ENTRY events since startup
`total OUT`	Total EXIT events since startup
`years`	Current year (`21` for 2021)
`month`	Current month
`day`	Current day (starting at 1)
`hour`	Current hour (00-23)
`minute`	Current minute
`second`	Current seconds
`report type`	`0xFF` for event counter, `0x20` on timeout

Protocol fields formats

The embedded presence sensor will be utilizing MQTT for communication. See more details

1. **Event Types:**
   - `diff IN`: Number of ENTRY events since the last sent attempt.
   - `diff OUT`: Number of EXIT events since the last sent attempt.
   - `total IN`: Total ENTRY events since startup.
   - `total OUT`: Total EXIT events since startup.
   - `years`: Current year (represented as '21' for 2021).
   - `month`: Current month.
   - `day`: Current day (starting at 1).
   - `hour`: Current hour (in 24-hour format).
   - `minute`: Current minute.
   - `second`: Current seconds.
   - `report type`: `0xFF` for event counter, `0x20` on timeout.
2. **Sending Conditions:**
   - A `_send_` action is triggered on the sensor when:
     - 5 events were counted (`report type = 0xFF`).
     - 30 minutes have passed since the last `_send_` (`report type = 0x20`).
3. **Important Notes:**
   - `total` values count all events since the start of the device and reset when they reach `65535`.
   - Missing messages: No resend if the broker is not reachable (due to network issues or service stops).
   - The controller (ARM) sending the message doesn't know the status of the sent operation by the MQTT controller (separate).
   - Date and time are obtained from the GPRS network since the sensor doesn't have a Real-Time Clock (RTC) to know when an event occurred.
   - A report is sent at device startup with all totals set to 0.
4. **Current Version Drawbacks:**

- The first two values (`diff IN` and `diff OUT`) are not working and are the same as totals, so they won't be used.
- The message length is 59 bytes, but only 14 are utilized.
- There's no Timezone (TZ) info, and the hive TZ will be used to compute time.
- Each value is incremented with 32, causing totals to reset to 0 when they overflow, reaching `65523`. Be cautious when looking for sensor restarts.

Recommended data types

Certainly, Cristi! Here are the recommended data types for each field, along with additional details on how they can be used:

1. `diff IN` **(Number of ENTRY events since last sent attempt):**
   - **Recommended Data Type:** Unsigned Integer
   - **Usage:** Track the number of entry events since the last successful data send. Useful for monitoring recent activity and changes.
2. `diff OUT` **(Number of EXIT events since last sent attempt):**
   - **Recommended Data Type:** Unsigned Integer
   - **Usage:** Similar to `diff IN`, it tracks the number of exit events since the last successful data send. Useful for monitoring recent exits.
3. `total IN` **(Total ENTRY events since startup):**
   - **Recommended Data Type:** Unsigned Integer
   - **Usage:** Represents the cumulative count of entry events since the device startup. Useful for maintaining a long-term record of all entry events.
4. `total OUT` **(Total EXIT events since startup):**
   - **Recommended Data Type:** Unsigned Integer
   - **Usage:** Similar to `total IN`, it represents the cumulative count of exit events since the device startup. Useful for maintaining a long-term record of all exit events.
5. `years` **(Current year):**
   - **Recommended Data Type:** Two-digit Integer
   - **Usage:** Represents the current year, such as '21' for 2021. Useful for timestamping events and tracking data over different years.
6. `month` **(Current month):**

- **Recommended Data Type:** Unsigned Integer (1-12)
- **Usage:** Represents the current month. Useful for organizing and analyzing data on a monthly basis.

7. `day` **(Current day):**
   - **Recommended Data Type:** Unsigned Integer (1-31)
   - **Usage:** Represents the current day of the month. Useful for daily tracking and analysis.
8. `hour` **(Current hour):**
   - **Recommended Data Type:** Unsigned Integer (0-23)
   - **Usage:** Represents the current hour in a 24-hour format. Useful for timestamping events and analyzing data based on the time of day.
9. `minute` **(Current minute):**
   - **Recommended Data Type:** Unsigned Integer (0-59)
   - **Usage:** Represents the current minute. Useful for more granular timestamping and analysis.
10. `second` **(Current seconds):**
    - **Recommended Data Type:** Unsigned Integer (0-59)
    - **Usage:** Represents the current seconds. Useful for precise timestamping and event timing.
11. `report type` **(Report type indicator):**
    - **Recommended Data Type:** Byte (or Enum with values 0xFF and 0x20)
    - **Usage:** Indicates the type of report being sent. ‘0xFF’ may signify an event counter, while ‘0x20’ may indicate a timeout. Useful for distinguishing between different types of reports.

These data types and details provide a foundation for implementing and interpreting the information transmitted by your embedded presence sensor.

Proposed improvements

To fix the above drawbacks:

- add a version number

  - as the protocol updates it is needed to correctly parse the future version of the report message

- add a message counter
  - allow to faster spotting of missed messages and signal bad connections to customers (so they can move the sensor)
- flag the restart message
  - *report type* = `0x21` when the device starts
  - will eliminate the overflow problem and make restart faster to identify
- add TZ field

Additional new messages

The following messages would be useful to configure the device and have a status update from it.

Status message

Sent by sensor at predefined intervals.

Bit	+0..7	+8..15	+16..23	+24..31
0	startup time			
32	battery state		sensor state	
64	discarded events		GPRS signal stregth	
96	firmware version			
128	configuration id			

Figure 3. status message

- length : 20 bytes
- configuration id is the last config id read from configuration topic
- is sent periodically based on the status report interval (defined in configuration)
- *static distance* the distance that is read statically by the sensor in that moment; should we send this instead of sensor status ?

Configuration

Sent by system, using `commands/sensor/<sensor-id>` topic.

Bit	+0..7	+8..15	+16..23	+24..31
0	sensor threshold cm		reporting interval min	
32	status interval min		event number threshold	
64	configuration id			

Figure 4. configuration

- length : 12 bytes
- read from `/config/...` topic after status report
- changes the sensor behavior and prompts an immediate status update

Report message

Bit	+0..7	+8..15	+16..23	+24..31
0	event type	sensor counter		
32	event timestamp in seconds			

The protocol outlines the structure for reporting presence, and it consist of two parts: the first 32 bits (4 bytes) represent the event details, and the next 32 bits (4 bytes) represent the event timestamp. Let’s break it down:

1. **Event Details (Bits 0-31):**
   - **Bits 0-7 (1 byte):** Event Type
     - Represents the type of event.
     - 0: Exit
     - 1: Entry
     - 2: Status (other types may be defined later)
   - **Bits 8-23 (2 bytes):** Distance in mm
     - Represents the distance associated with the event, measured in millimeters.
   - **Bits 24-31 (1 byte):** Reserved

- Currently not specified; reserved for potential future use.

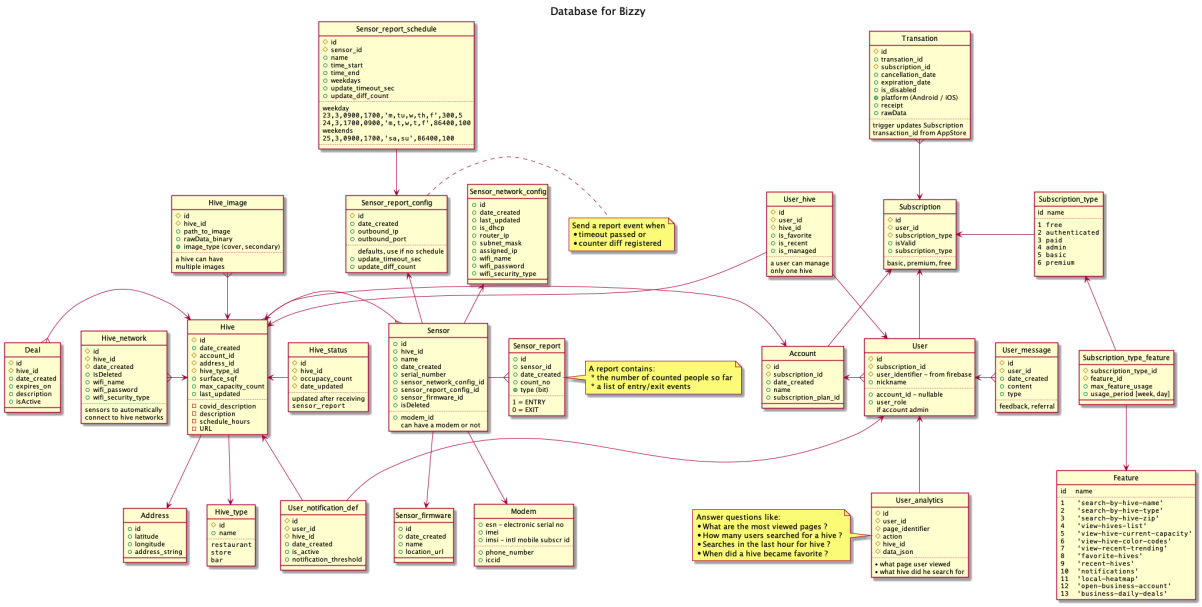
2. **Event Timestamp (Bits 32-63):**
   - **Bits 32-63 (4 bytes):** Event Timestamp in Seconds
     - Represents the timestamp of the event, indicating when it occurred, measured in seconds.
3. **Event Type:**
   - The event type specifies whether it’s an entry, exit, or another type of event (e.g., status).
   - This information is crucial for understanding the nature of the reported presence.
4. **Distance in mm:**
   - Provides the distance associated with the event, measured in millimeters.
   - This could be the distance from a sensor or some relevant metric depending on the application.
5. **Event Timestamp:**
   - Represents the time at which the event occurred.
   - Measured in seconds, this timestamp helps in tracking the chronological order of events.
6. **Status Type (Reserved):**
   - The protocol reserves a byte for potential future use (Bits 24-31).
   - Its purpose is not currently specified, leaving room for expansion or modification in the protocol.
7. **Event Type Enumeration:**
   - A legend at the end of the protocol specifies the mapping of numeric values to event types.

This protocol provides a concise and structured way to report presence events, including essential details such as event type, distance, and timestamp. The reserved field allows for potential future extensions without modifying the existing structure.

Implementing this protocol involves parsing the received bits to extract relevant information based on the defined structure.

Database

Database diagram



Major components

Major components of the database are described below:

1. **User-related Entities:**
   - **User:** Represents users with attributes like `id`, `nickname`, `subscription_id`, and `user_role`.
   - **User_message:** Stores user messages with `id`, `user_id`, `date_created`, `content`, and `type`.
   - **User_analytics:** Captures user analytics with details like `id`, `user_id`, `page_identifier`, `action`, and `data_json`.
2. **Subscription Entities:**
   - **Subscription:** Manages user subscriptions with attributes such as `id`, `user_id`, and `subscription_type`.
   - **Subscription_type:** Defines various subscription types like basic, premium, etc.
   - **Transaction:** Represents transactions related to subscriptions.
3. **Feature-related Entities:**
   - **Feature:** Lists features with `id` and `name`.

- **User_hive:** Contains information about users' interaction with hives, including favorites and recent hives.
- **User_notification_def:** Manages user notification definitions.

4. **Location and Address Entities:**
   - **Address:** Represents geographical addresses.
   - **Account:** Contains account details with attributes like `id`, `subscription_id`, and `name`.
5. **Hive-related Entities:**
   - **Hive:** Describes hives with information such as `id`, `date_created`, `account_id`, `address_id`, and `hive_type_id`.
   - **Hive_status:** Keeps track of hive occupancy status.
   - **Hive_network:** Manages hive network details.
   - **Hive_image:** Stores images related to hives.
6. **Deal Entity:**
   - **Deal:** Contains details about deals related to hives.
7. **Hive Type Entity:**
   - **Hive_type:** Classifies hives into types like restaurant, store, bar, etc.
8. **Sensor and Sensor-related Entities:**
   - **Sensor:** Represents sensors with attributes like `id`, `hive_id`, `name`, etc.
   - **Sensor_report:** Stores sensor reports with details like `id`, `sensor_id`, `date_created`, `count_no`, and `type`.
   - **Modem:** Represents modem details for sensors.
   - **Sensor_network_config:** Manages sensor network configurations.
   - **Sensor_report_config:** Handles sensor report configurations.
   - **Sensor_report_schedule:** Defines schedules for sensor reports.

This schema is comprehensive and covers aspects such as user management, subscriptions, features, hives, deals, and sensor-related functionalities. It provides a foundation for building a system with real-time analytics, user interactions, and IoT sensor data. The notes in the script provide additional context for certain entities and their functionalities.

More details about data relations:

1. **User-related Entities:**
   - **User <¬ User_message:** Users can send messages, establishing a one-to-many relationship. One user can have multiple messages, but each message belongs to a single user.
   - **User ¬down- User_analytics:** Users are associated with analytics data, indicating a one-to-many relationship. Each user can have multiple analytics records.
2. **Subscription Entities:**
   - **User -up¬ Subscription:** Each user can have one or more subscriptions, creating a one-to-many relationship. However, a subscription is linked to a single user.
   - **Account -up¬ Subscription:** An account can have multiple subscriptions, forming a one-to-many relationship. However, each subscription is linked to a single account.
   - **Transaction }-down¬ Subscription:** Transactions are related to subscriptions, establishing a one-to-many relationship. A transaction is associated with a specific subscription.
3. **Feature-related Entities:**
   - **Subscription_type ¬down- Subscription_type_feature ¬ Feature:** Describes the relationship between subscription types and features. Each subscription type can have multiple features, and each feature can be associated with multiple subscription types.
   - **User ¬up- User_hive ¬ Hive:** Represents the relationship between users and hives. Users can interact with multiple hives, and each hive can be associated with multiple users.
   - **User_notification_def -up¬ User:** Users can have multiple notification definitions, creating a one-to-many relationship. Each definition is linked to a specific user.
4. **Location and Address Entities:**
   - **Hive -down¬ Address:** Each hive is associated with a specific address, forming a one-to-one relationship.
   - **Account ¬left-\{ Hive:** Accounts are related to hives, indicating a one-to-many relationship. An account can be associated with multiple hives, but each hive is linked to a single account.
5. **Hive-related Entities:**
   - **Hive -down¬ Hive_type:** Describes the relationship between hives and hive types. Each hive is of a specific type, establishing a one-to-one relationship.
   - **Hive ¬\{ Sensor:** Hives can have multiple sensors, indicating a one-to-many relationship. Each sensor is associated with a specific hive.

- **Sensor -up¬ Sensor_network_config:** Describes the relationship between sensors and their network configurations. Each sensor has a specific network configuration.
- **Sensor -up¬ Sensor_report_config:** Represents the relationship between sensors and report configurations. Each sensor has a specific report configuration.
- **Sensor -right-\{ Sensor_report:** Indicates the relationship between sensors and their reports. Each sensor can have multiple reports, creating a one-to-many relationship.

6. **Sensor-related Entities:**
   - **Sensor -down¬ Modem:** Describes the relationship between sensors and modems. Each sensor can have a modem.
7. **Miscellaneous Entities:**
   - **Hive_status -left¬ Hive:** Hive status is associated with a specific hive, indicating a one-to-one relationship.
   - **Hive_network }-right¬ Hive:** Describes the relationship between hive networks and hives. Each hive can have multiple network configurations.
   - **Hive_image ¬up-\{ Hive:** Represents the relationship between hive images and hives. Each hive can have multiple images.
   - **Deal }-right¬ Hive:** Indicates that deals are associated with specific hives, forming a one-to-many relationship.

These detailed descriptions should provide a clearer understanding of how the various entities in the database are connected.

App features

Features outline

The database schema outlined provides a rich set of features for an application related to the Bizzy service. Here are some potential features that will be implemented based on this database:

1. **User Management:**
   - **User Profiles:** Allow users to create and manage profiles with details like nickname, subscription information, and roles.
   - **Messaging System:** Implement a messaging system that enables users to send and receive messages.
   - **User Analytics:** Provide insights into user activities, such as most viewed pages, hive searches, and favorites.

2. **Subscription and Account Management:**
   - **Subscription Plans:** Offer different subscription plans (basic, premium, free) with corresponding features.
   - **Transaction Tracking:** Keep track of subscription transactions, including cancellation and expiration dates.
   - **Account Features:** Enable features related to account management, such as updating account details.
3. **Hive and Location Services:**
   - **Hive Listings:** Display a list of hives with details like type, capacity, and last updated information.
   - **Hive Details:** Allow users to view detailed information about a specific hive, including images and deals.
   - **Hive Search:** Implement search functionality based on hive name, type, ZIP code, etc.
   - **Occupancy Tracking:** Display real-time occupancy status for hives based on sensor reports.
4. **User Interaction with Hives:**
   - **Favorite Hives:** Allow users to mark hives as favorites.
   - **Recent Hives:** Display recently visited hives for each user.
   - **User Notifications:** Provide notification settings for users based on their preferences.
5. **Deals and Notifications:**
   - **Deal Listings:** Show active deals associated with hives.
   - **Deal Expiration Notifications:** Notify users when deals are about to expire.
   - **Custom Notifications:** Allow users to set notification thresholds for specific hives.
6. **Sensor Integration:**
   - **Sensor Data Display:** Show sensor reports and analytics related to hives.
   - **Sensor Configuration:** Provide settings for configuring sensor networks and report schedules.
   - **Real-Time Updates:** Offer real-time updates on hive occupancy and sensor data.
7. **Image Gallery:**
   - **Hive Image Gallery:** Allow users to browse images related to each hive.

- **Image Types:** Differentiate between cover and secondary images for hives.

8. **Business and Analytics Features:**
   - **Business Account:** Enable businesses to open and manage their accounts.
   - **Daily Deals for Businesses:** Provide a platform for businesses to offer daily deals.
   - **Analytics Queries:** Support analytics queries to answer questions about user behavior and hive interactions.
9. **Location Services:**
   - **Geographical Information:** Utilize address and location data for mapping and navigation features.
   - **Location-Based Search:** Implement search features based on geographical proximity.
10. **Security and Permissions:**
    - **User Roles and Permissions:** Define roles such as account admin and set appropriate permissions.
    - **Authentication:** Implement secure authentication, especially considering the sensitive nature of user data.

These features cater to a diverse set of functionalities ranging from user interaction with hives to business account management and analytics. Implementation details would depend on the specific goals and requirements of the application.

Feature: Real-Time Hive Occupancy Monitoring

Description: This feature involves integrating sensors within hives to monitor real-time occupancy. The sensors count the number of people entering and exiting a hive, providing live data on how crowded or vacant a particular location is. This information can be valuable for users looking to choose a less crowded hive or for businesses to optimize their operations based on customer traffic.

Implementation Steps:

1. **Sensor Deployment:**
   - Install sensors equipped with the necessary firmware and configurations in strategic locations within each hive.
   - Configure the sensors to communicate with the central server/database.
2. **Sensor Data Collection:**
   - Sensors periodically send reports to the database, indicating the number of people entering and exiting the hive.

   - Sensor reports include timestamps, counts, and types (entry or exit).

3. **Database Integration:**
   - Store sensor reports in the `Sensor_report` table, associating each report with the corresponding sensor and hive.
   - Update the `Hive_status` table with the latest occupancy count and timestamp.
4. **Real-Time Updates:**
   - Implement a real-time update mechanism in the application to fetch and display the latest occupancy data for each hive.
   - Users can see live updates on how busy a hive is at any given moment.
5. **User Notifications:**
   - Allow users to set notification thresholds for hives based on occupancy levels.
   - Implement a notification system to alert users when a hive's occupancy surpasses or falls below their specified threshold.
6. **Occupancy Trends and Analytics:**
   - Utilize the `User_analytics` and other related tables to store and analyze historical occupancy data.
   - Provide users and businesses with insights into peak hours, daily trends, and popular hives.

User Interaction: - Users can access the application and view real-time occupancy status for hives. - Set preferences to receive notifications when a hive becomes too crowded or when a favorite hive has lower occupancy. - Access historical data and analytics to make informed decisions about when to visit specific hives.

Benefits: - Enhances user experience by providing valuable information on hive occupancy. - Allows businesses to optimize staffing and services based on real-time customer traffic. - Enables users to make data-driven decisions when choosing a hive to visit.

Feature: Hive Analytics Dashboard

Let's explore an example of a business and analytics feature for the Bizzy application: "Hive Analytics Dashboard."

Description: The Hive Analytics Dashboard provides businesses with a comprehensive overview of their hive-related data, allowing them to make informed decisions, optimize operations, and track performance metrics.

Implementation Steps:

1. **User Interface for Businesses:**
   - Develop a dedicated analytics dashboard accessible to businesses with the necessary permissions.
   - Include graphical representations and charts for easy visualization of key metrics.
2. **Key Performance Indicators (KPIs):**
   - Define and display essential KPIs such as total hive visits, average occupancy, peak hours, and customer demographics.
   - Utilize data from tables like `User_analytics` and `Hive_status` for these metrics.
3. **Trend Analysis:**
   - Implement tools to analyze trends in user behavior, such as popular hives, frequently searched hives, and changes in user preferences over time.
   - Use historical data stored in the `User_analytics` table for trend analysis.
4. **Peak Hour Prediction:**
   - Develop a predictive model to estimate peak hours for each hive based on historical data.
   - Use machine learning algorithms or statistical methods to forecast when a hive is likely to experience high foot traffic.
5. **Business-specific Metrics:**
   - Allow businesses to define and track custom metrics based on their specific goals. For example, monitor the success of a promotional campaign or track the impact of a new deal.
6. **Comparative Analysis:**
   - Enable businesses to compare the performance of different hives or analyze the impact of changes in hive features on user engagement.
   - Implement features to select specific time periods for comparison.
7. **User Feedback Analysis:**
   - Integrate user feedback data from the `User_message` table into the analytics dashboard.
   - Provide sentiment analysis tools to understand customer satisfaction and identify areas for improvement.

User Interaction: - Businesses log in to the analytics dashboard to access a visual

representation of key metrics and trends. - Receive automated insights and recommendations based on the analytics data. - Make data-driven decisions to optimize hive operations, marketing strategies, and customer experiences.

Benefits: - Empowers businesses with actionable insights for strategic decision-making. - Helps businesses understand customer behavior, preferences, and satisfaction levels. - Optimizes resource allocation, staffing, and marketing efforts based on data-driven analysis.

This Hive Analytics Dashboard feature provides businesses with a powerful tool to leverage the collected data, fostering continuous improvement and informed decision-making.

!!! Feature: Deal Notifications for Users

Let’s explore an example of the “Deals and Notifications” feature for the Bizzy application: “Deal Notifications for Users.”

Description: This feature involves notifying users about active deals related to hives they have shown interest in or have favorited. Users receive timely notifications to encourage engagement and take advantage of special deals offered by businesses.

Implementation Steps:

1. **Deal Management:**
   - Businesses create and manage deals associated with specific hives.
   - Deals include details such as a description, expiration date, and terms.
2. **User Deal Preferences:**
   - Allow users to set deal preferences, such as the types of deals they are interested in (e.g., discounts, promotions) and notification thresholds.
3. **Notification Triggers:**
   - Implement a notification system that triggers notifications based on specific events, such as the availability of a new deal, deal expiration, or when a favorite hive offers a deal.
4. **Deal Expiry Alerts:**
   - Notify users in advance when a deal is about to expire, encouraging them to take advantage of the offer before it ends.
   - Use data from the `Deal` table to determine deal expiration dates.
5. **Favorite Hive Deals:**
   - Users receive notifications when a favorite hive introduces a new deal or has an

ongoing promotion.

- Utilize the `User_hive` and `Deal` tables to determine which hives the user has favorited and if there are associated deals.

6. **Deal Recommendations:**
   - Provide personalized deal recommendations based on user preferences, historical engagement data, and trending deals.
   - Use data from the `User_analytics` table to understand user behavior and preferences.

User Interaction: - Users receive push notifications or in-app notifications about new deals, expiring deals, or promotions from their favorite hives. - Users can click on notifications to view detailed information about the deal and quickly access the associated hive.

Benefits: - Enhances user engagement by providing personalized and timely deal notifications. - Encourages users to visit hives and take advantage of special offers. - Fosters a positive user experience by delivering relevant and valuable information.

This “Deal Notifications for Users” feature creates a dynamic and engaging experience for users, encouraging them to explore hives, discover deals, and participate in promotions offered by businesses on the Bizzy platform.

Queue engine module

Messaging systems work basically as instant messaging for applications. An application decides to communicate an event to another application (or multiple applications), it assembles the data to be sent, hits the “send” button and there we go—the messaging system takes care of the rest.

Unlike instant messaging, though, messaging systems have no GUI and assume no human beings at the endpoints capable of intelligent intervention when something goes wrong. Messaging systems thus have to be both fault-tolerant and much faster than common instant messaging.

Application

Our primary concern at the time was with the performance: If there’s a server in the middle, each message has to pass the network twice (from the sender to the broker and from the broker to the receiver) inducing a penalty in terms of both latency and throughput. Moreover, if all the messages are passed through the broker, at some point it’s bound to become the bottleneck.

A secondary concern was related to large-scale deployments: when the deployment crosses organisational boundaries the concept of a central authority managing the whole message flow doesn’t apply any more. No company is willing to cede control to a server in different company; there are trade secrets and there’s legal liability. The result in practice is that there’s one messaging server per company, with hand-written bridges to connect it to messaging systems in other companies. The whole ecosystem is thus heavily fragmented, and maintaining a large number of bridges for every company involved doesn’t make the situation better. To solve this problem, we need a fully distributed architecture, an architecture where every component can be possibly governed by a different business entity. Given that the unit of management in server-based architecture is the server, we can solve the problem by installing a separate server for each component. In such a case we can further optimize the design by making the server and the component share the same processes. What we end up with is a messaging library.

ØMQ was started when we got an idea about how to make messaging work without a central server. It required turning the whole concept of messaging upside down and replacing the model of an autonomous centralised store of messages in the center of the network with a “smart endpoint, dumb network” architecture based on the end-to-end principle. The technical consequence of that decision was that ØMQ, from the very beginning, was a library, not an application.

In the meantime we've been able to prove that this architecture is both more efficient (lower latency, higher throughput) and more flexible (it's easy to build arbitrary complex topologies instead of being tied to classic hub-and-spoke model).

One of the unintended consequences, however, was that opting for the library model improved the usability of the product. Over and over again users express their happiness about the fact that they don't have to install and manage a stand-alone messaging server. It turns out that not having a server is a preferred option as it cuts operational cost (no need to have a messaging server admin) and improves time-to-market (no need to negotiate the need to run the server with the client, the management or the operations team).

The lesson learned is that when starting a new project, you should opt for the library design if at all possible. It's pretty easy to create an application from a library by invoking it from a trivial program; however, it's almost impossible to create a library from an existing executable. A library offers much more flexibility to the users, at the same time sparing them non-trivial administrative effort.

Global State

Global variables don't play well with libraries. A library may be loaded several times in the process but even then there's only a single set of global variables.

To prevent this problem, the ØMQ library has no global variables. Instead, a user of the library is responsible for creating the global state explicitly.

The object containing the global state is called *context*. While from the user's perspective context looks more or less like a pool of worker threads, from ØMQ's perspective it's just an object to store any global state that we happen to need. In the picture above, `libA` would have its own context and `libB` would have its own as well. There would be no way for one of them to break or subvert the other one.

The lesson here is pretty obvious: Don't use global state in libraries. If you do, the library is likely to break when it happens to be instantiated twice in the same process.

Performance

Which metric should we focus on? What's the relationship between the two? Isn't it obvious? Run the test, divide the overall time of the test by number of messages passed and what you get is latency. Divide the number of messages by time and what you get is throughput. In other words, latency is the inverse value of throughput. Trivial, right?

Instead of starting coding straight away we spent some weeks investigating the performance

metrics in detail and we found out that the relationship between throughput and latency is much more subtle than that, and often the metrics are quite counter-intuitive.

Imagine A sending messages to B.The overall time of the test is 6 seconds. There are 5 messages passed. Therefore the throughput is 0.83 msgs/sec (5/6) and the latency is 1.2 sec (6/5), right?

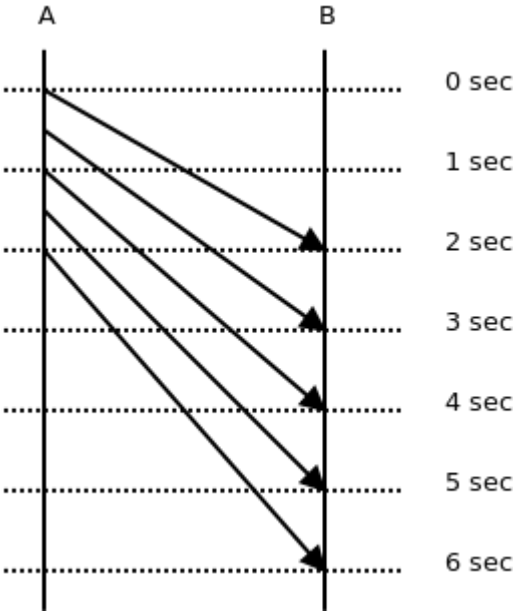


Figure 24.2: Sending messages from A to B

Have a look at the diagram again. It takes a different time for each message to get from A to B: 2 sec, 2.5 sec, 3 sec, 3.5 sec, 4 sec. The average is 3 seconds, which is pretty far away from our original calculation of 1.2 second. This example shows the misconceptions people are intuitively inclined to make about performance metrics.

Now have a look at the throughput. The overall time of the test is 6 seconds. However, at A it takes just 2 seconds to send all the messages. From A's perspective the throughput is 2.5 msgs/sec (5/2). At B it takes 4 seconds to receive all messages. So from B's perspective the throughput is 1.25 msgs/sec (5/4). Neither of these numbers matches our original calculation of 1.2 msgs/sec.

To make a long story short, latency and throughput are two different metrics; that much is obvious. The important thing is to understand the difference between the two and their mutual relationship. Latency can be measured only between two different points in the system; There's no such thing as latency at point A. Each message has its own latency. You can average the latencies of multiple messages; however, there's no such thing as latency of a stream of messages.

Throughput, on the other hand, can be measured only at a single point of the system. There's

a throughput at the sender, there's a throughput at the receiver, there's a throughput at any intermediate point between the two, but there's no such thing as overall throughput of the whole system. And throughput make sense only for a set of messages; there's no such thing as throughput of a single message.

As for the relationship between throughput and latency, it turns out there really is a relationship; however, the formula involves integrals and we won't discuss it here. For more information, read the literature on queueing theory.

There are many more pitfalls in benchmarking the messaging systems that we won't go further into. The stress should rather be placed on the lesson learned: Make sure you understand the problem you are solving. Even a problem as simple as "make it fast" can take lot of work to understand properly. What's more, if you don't understand the problem, you are likely to build implicit assumptions and popular myths into your code, making the solution either flawed or at least much more complex or much less useful than it could possibly be.

Critical Path

We discovered during the optimization process that three factors have a crucial impact on performance:

- Number of memory allocations
- Number of system calls
- Concurrency model

However, not every memory allocation or every system call has the same effect on performance. The performance we are interested in in messaging systems is the number of messages we can transfer between two endpoints during a given amount of time. Alternatively, we may be interested in how long it takes for a message to get from one endpoint to another.

However, given that ØMQ is designed for scenarios with long-lived connections, the time it takes to establish a connection or the time needed to handle a connection error is basically irrelevant. These events happen very rarely and so their impact on overall performance is negligible.

The part of a codebase that gets used very frequently, over and over again, is called the *critical path*; optimization should focus on the critical path.

Let's have a look at an example: ØMQ is not extremely optimized with respect to memory allocations. For example, when manipulating strings, it often allocates a new string for each

intermediate phase of the transformation. However, if we look strictly at the critical path—the actual message passing—we'll find out that it uses almost no memory allocations. If messages are small, it's just one memory allocation per 256 messages (these messages are held in a single large allocated memory chunk). If, in addition, the stream of messages is steady, without huge traffic peaks, the number of memory allocations on the critical path drops to zero (the allocated memory chunks are not returned to the system, but re-used over and over again).

Lesson learned: optimize where it makes difference. Optimizing pieces of code that are not on the critical path is wasted effort.

Allocating Memory

Assuming that all the infrastructure was initialised and a connection between two endpoints has been established, there's only one thing to allocate when sending a message: the message itself. Thus, to optimize the critical path we had to look into how messages are allocated and passed up and down the stack.

It's common knowledge in the high-performance networking field that the best performance is achieved by carefully balancing the cost of message allocation and the cost of message copying (for example, http://hal.inria.fr/docs/00/29/28/31/PDF/Open-MX-IOAT.pdf: see different handling of "small", "medium" and "large" messages). For small messages, copying is much cheaper than allocating memory. It makes sense to allocate no new memory chunks at all and instead to copy the message to preallocated memory whenever needed. For large messages, on the other hand, copying is much more expensive than memory allocation. It makes sense to allocate the message once and pass a pointer to the allocated block, instead of copying the data. This approach is called "zero-copy".

ØMQ handles both cases in a transparent manner. A ØMQ message is represented by an opaque handle. The content of very small messages is encoded directly in the handle. So making a copy of the handle actually copies the message data. When the message is larger, it's allocated in a separate buffer and the handle contains just a pointer to the buffer. Making a copy of the handle doesn't result in copying the message data, which makes sense when the message is megabyteslong. It should be noted that in the latter case the buffer is reference-counted so that it can be referenced by multiple handles without the need to copy the data.

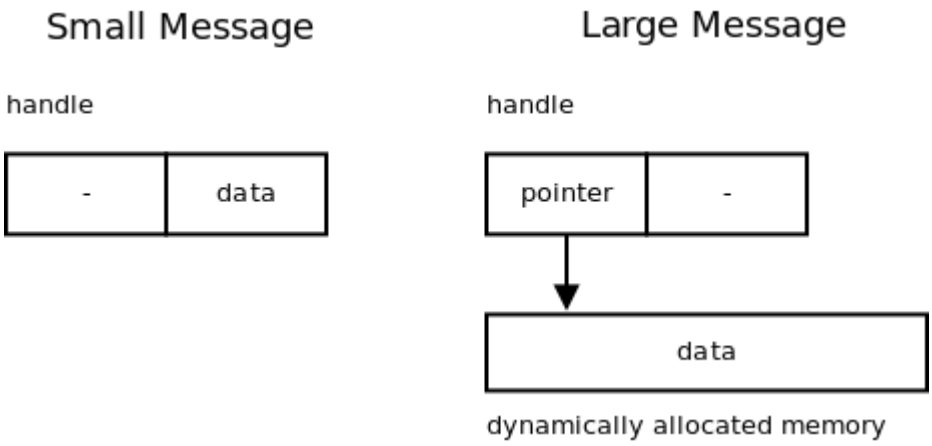


Figure 24.3: Message copying (or not)

Lesson learned: When thinking about performance, don't assume there's a single best solution. It may happen that there are several subclasses of the problem (e.g., small messages vs. large messages), each having its own optimal algorithm.

Batching

It has already been mentioned that the sheer number of system calls in a messaging system can result in a performance bottleneck. Actually, the problem is much more generic than that. There's a non-trivial performance penalty associated with traversing the call stack and thus, when creating high-performance applications, it's wise to avoid as much stack traversing as possible.

Consider Figure 24.4. To send four messages, you have to traverse the entire network stack four times (i.e., ØMQ, glibc, user/kernel space boundary, TCP implementation, IP implementation, Ethernet layer, the NIC itself and back up the stack again).

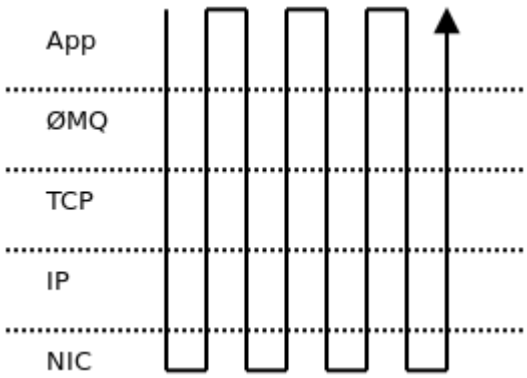


Figure 24.4: Sending four messages

However, if you decide to join those messages into a single batch, there would be only one traversal of the stack (Figure 24.5). The impact on message throughput can be overwhelming: up to two orders of magnitude, especially if the messages are small and hundreds of them can

be packed into a single batch.

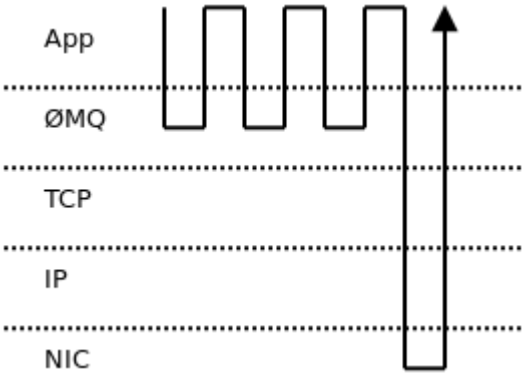


Figure 24.5: Batching messages

On the other hand, batching can have negative impact on latency. Let's take, for example, the well-known Nagle's algorithm, as implemented in TCP. It delays the outbound messages for a certain amount of time and merges all the accumulated data into a single packet. Obviously, the end-to-end latency of the first message in the packet is much worse than the latency of the last one. Thus, it's common for applications that need consistently low latency to switch Nagle's algorithm off. It's even common to switch off batching on all levels of the stack (e.g., NIC's interrupt coalescing feature).

But again, no batching means extensive traversing of the stack and results in low message throughput. We seem to be caught in a throughput versus latency dilemma.

ØMQ tries to deliver consistently low latencies combined with high throughput using the following strategy: when message flow is sparse and doesn't exceed the network stack's bandwidth, ØMQ turns all the batching off to improve latency. The trade-off here is somewhat higher CPU usage—we still have to traverse the stack frequently. However, that isn't considered to be a problem in most cases.

When the message rate exceeds the bandwidth of the network stack, the messages have to be queued—stored in memory till the stack is ready to accept them. Queueing means the latency is going to grow. If the message spends one second in the queue, end-to-end latency will be at least one second. What's even worse, as the size of the queue grows, latencies will increase gradually. If the size of the queue is not bound, the latency can exceed any limit.

It has been observed that even though the network stack is tuned for lowest possible latency (Nagle's algorithm switched off, NIC interrupt coalescing turned off, etc.) latencies can still be dismal because of the queueing effect, as described above.

In such situations it makes sense to start batching aggressively. There's nothing to lose as the latencies are already high anyway. On the other hand, aggressive batching improves

throughput and can empty the queue of pending messages—which in turn means the latency will gradually drop as the queueing delay decreases. Once there are no outstanding messages in the queue, the batching can be turned off to improve the latency even further.

One additional observation is that the batching should only be done on the topmost level. If the messages are batched there, the lower layers have nothing to batch anyway, and so all the batching algorithms underneath do nothing except introduce additional latency.

Lesson learned: To get optimal throughput combined with optimal response time in an asynchronous system, turn off all the batching algorithms on the low layers of the stack and batch on the topmost level. Batch only when new data are arriving faster than they can be processed.

Queue architecture Overview

Up to this point we have focused on generic principles that make ØMQ fast. From now on we'll have a look at the actual architecture of the system (Figure 24.6).

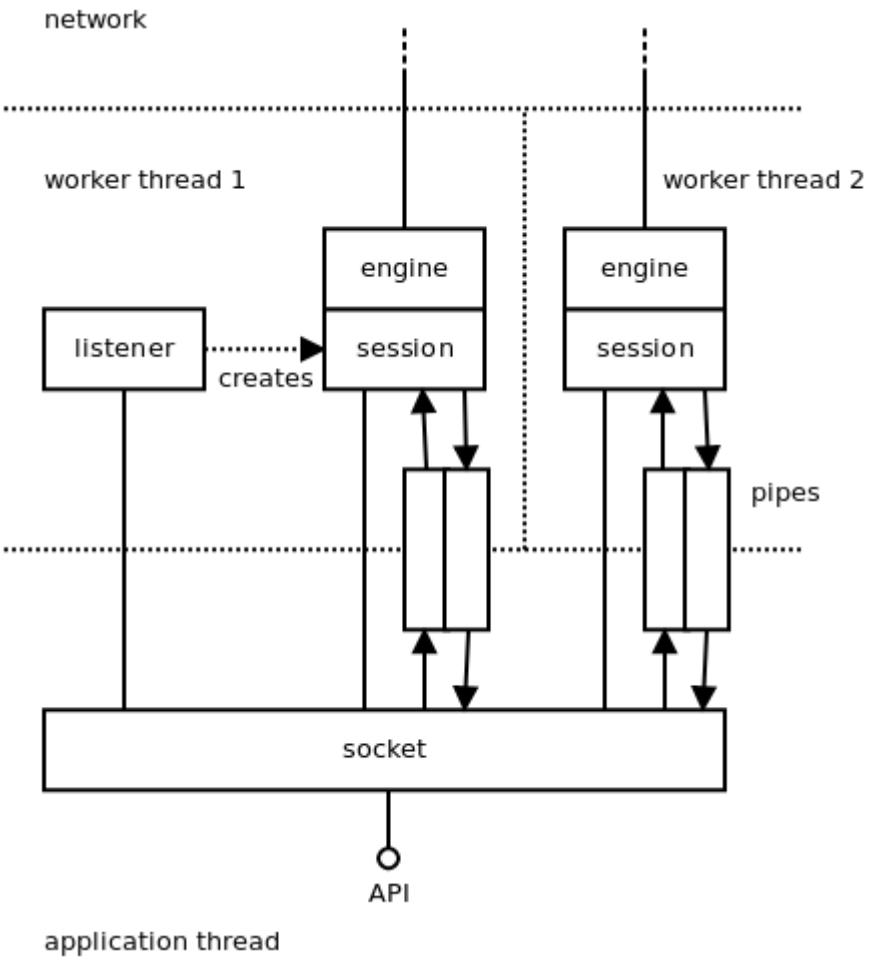


Figure 24.6: architecture

The user interacts with ØMQ using so-called "sockets". They are pretty similar to TCP

sockets, the main difference being that each socket can handle communication with multiple peers, a bit like unbound UDP sockets do.

The socket object lives in the user's thread (see the discussion of threading models in the next section). Aside from that, ØMQ is running multiple worker threads that handle the asynchronous part of the communication: reading data from the network, enqueueing messages, accepting incoming connections, etc.

There are various objects living in the worker threads. Each of these objects is owned by exactly one parent object (ownership is denoted by a simple full line in the diagram). The parent can live in a different thread than the child. Most objects are owned directly by sockets; however, there are couple of cases where an object is owned by an object which is owned by the socket. What we get is a tree of objects, with one such tree per socket. The tree is used during shut down; no object can shut itself down until it closes all its children. This way we can ensure that the shut down process works as expected; for example, that pending outbound messages are pushed to the network prior to terminating the sending process.

Roughly speaking, there are two kinds of asynchronous objects; there are objects that are not involved in message passing and there are objects that are. The former have to do mainly with connection management. For example, a TCP listener object listens for incoming TCP connections and creates an engine/session object for each new connection. Similarly, a TCP connector object tries to connect to the TCP peer and when it succeeds it creates an engine/session object to manage the connection. When such connection fails, the connector object tries to re-establish it.

The latter are objects that are handling data transfer itself. These objects are composed of two parts: the *session object* is responsible for interacting with the ØMQ socket, and the *engine object* is responsible for communication with the network. There's only one kind of the session object, but there's a different engine type for each underlying protocol ØMQ supports. Thus, we have TCP engines, IPC (inter-process communication) engines, PGM engines (a reliable multicast protocol, see RFC 3208), etc. The set of engines is extensible—in the future we may choose to implement, say, a WebSocket engine or an SCTP engine.

The sessions are exchanging messages with the sockets. There are two directions to pass messages in and each direction is handled by a pipe object. Each pipe is basically a lock-free queue optimized for fast passing of messages between threads.

Finally, there's a context object (discussed in the previous sections but not shown on the diagram) that holds the global state and is accessible by all the sockets and all the asynchronous objects.

Concurrency Model

One of the requirements for ØMQ was to take advantage of multi-core boxes; in other words, to scale the throughput linearly with the number of available CPU cores.

Our previous experience with messaging systems showed that using multiple threads in a classic way (critical sections, semaphores, etc.) doesn't yield much performance improvement. In fact, a multi-threaded version of a messaging system can be slower than a single-threaded one, even if measured on a multi-core box. Individual threads are simply spending too much time waiting for each other while, at the same time, eliciting a lot of context switching that slows the system down.

Given these problems, we've decided to go for a different model. The goal was to avoid locking entirely and let each thread run at full speed. The communication between threads was to be provided via asynchronous messages (events) passed between the threads. This, as insiders know, is the classic *actor model*.

The idea was to launch one worker thread per CPU core—having two threads sharing the same core would only mean a lot of context switching for no particular advantage. Each internal ØMQ object, such as say, a TCP engine, would be tightly bound to a particular worker thread. That, in turn, means that there's no need for critical sections, mutexes, semaphores and the like. Additionally, these ØMQ objects won't be migrated between CPU cores so would thus avoid the negative performance impact of cache pollution

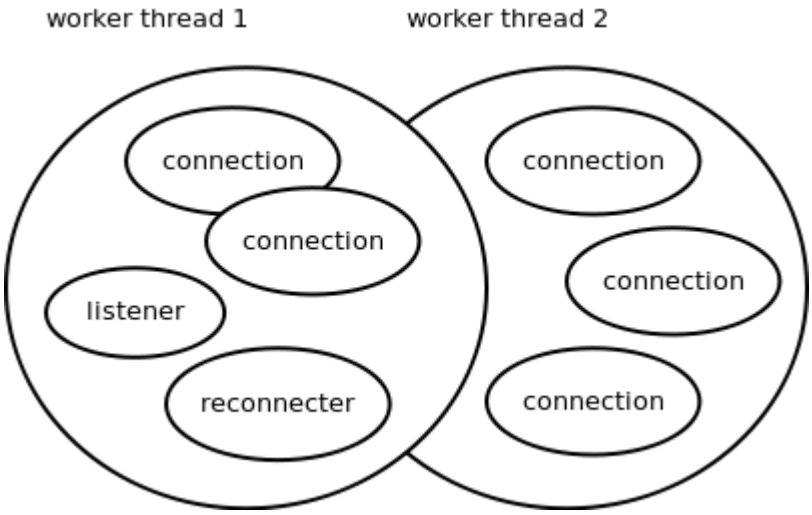


Figure 24.7: Multiple worker threads

This design makes a lot of traditional multi-threading problems disappear. Nevertheless, there's a need to share the worker thread among many objects, which in turn means there has to be some kind of cooperative multitasking. This means we need a scheduler; objects need to be event-driven rather than being in control of the entire event loop; we have to take care of

arbitrary sequences of events, even very rare ones; we have to make sure that no object holds the CPU for too long; etc.

In short, the whole system has to become fully asynchronous. No object can afford to do a blocking operation, because it would not only block itself but also all the other objects sharing the same worker thread. All objects have to become, whether explicitly or implicitly, state machines. With hundreds or thousands of state machines running in parallel you have to take care of all the possible interactions between them and—most importantly—of the shutdown process.

It turns out that shutting down a fully asynchronous system in a clean way is a dauntingly complex task. Trying to shut down a thousand moving parts, some of them working, some idle, some in the process of being initiated, some of them already shutting down by themselves, is prone to all kinds of race conditions, resource leaks and similar. The shutdown subsystem is definitely the most complex part of ØMQ. A quick check of the bug tracker indicates that some 30–50% of reported bugs are related to shutdown in one way or another.

Lesson learned: When striving for extreme performance and scalability, consider the actor model; it's almost the only game in town in such cases. However, if you are not using a specialised system like Erlang or ØMQ itself, you'll have to write and debug a lot of infrastructure by hand. Additionally, think, from the very beginning, about the procedure to shut down the system. It's going to be the most complex part of the codebase and if you have no clear idea how to implement it, you should probably reconsider using the actor model in the first place.

Lock-Free Algorithms

Lock-free algorithms have been in vogue lately. They are simple mechanisms for inter-thread communication that don't rely on the kernel-provided synchronisation primitives, such as mutexes or semaphores; rather, they do the synchronisation using atomic CPU operations, such as atomic compare-and-swap (CAS). It should be understood that they are not literally lock-free—instead, locking is done behind the scenes on the hardware level.

ØMQ uses a lock-free queue in pipe objects to pass messages between the user's threads and ØMQ's worker threads. There are two interesting aspects to how ØMQ uses the lock-free queue.

First, each queue has exactly one writer thread and exactly one reader thread. If there's a need for 1-to-`N` communication, multiple queues are created (Figure 24.8). Given that this way the queue doesn't have to take care of synchronising the writers (there's only one writer) or readers (there's only one reader) it can be implemented in an extra-efficient way.

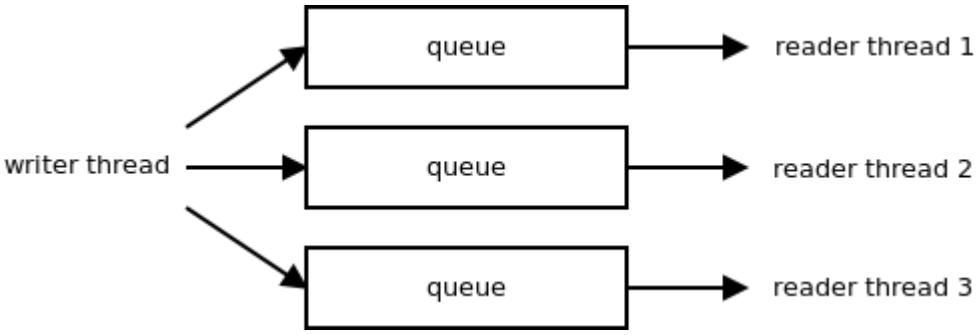


Figure 24.8: Queues

Second, we realised that while lock-free algorithms were more efficient than classic mutex-based algorithms, atomic CPU operations are still rather expensive (especially when there's contention between CPU cores) and doing an atomic operation for each message written and/or each message read was slower than we were willing to accept.

The way to speed it up—once again—was batching. Imagine you had 10 messages to be written to the queue. It can happen, for example, when you received a network packet containing 10 small messages. Receiving a packet is an atomic event; you cannot get half of it. This atomic event results in the need to write 10 messages to the lock-free queue. There's not much point in doing an atomic operation for each message. Instead, you can accumulate the messages in a "pre-write" portion of the queue that's accessed solely by the writer thread, and then flush it using a single atomic operation.

The same applies to reading from the queue. Imagine the 10 messages above were already flushed to the queue. The reader thread can extract each message from the queue using an atomic operation. However, it's overkill; instead, it can move all the pending messages to a "pre-read" portion of the queue using a single atomic operation. Afterwards, it can retrieve the messages from the "pre-read" buffer one by one. "Pre-read" is owned and accessed solely by the reader thread and thus no synchronisation whatsoever is needed in that phase.

The arrow on the left of Figure 24.9 shows how the pre-write buffer can be flushed to the queue simply by modifying a single pointer. The arrow on the right shows how the whole content of the queue can be shifted to the pre-read by doing nothing but modifying another pointer.

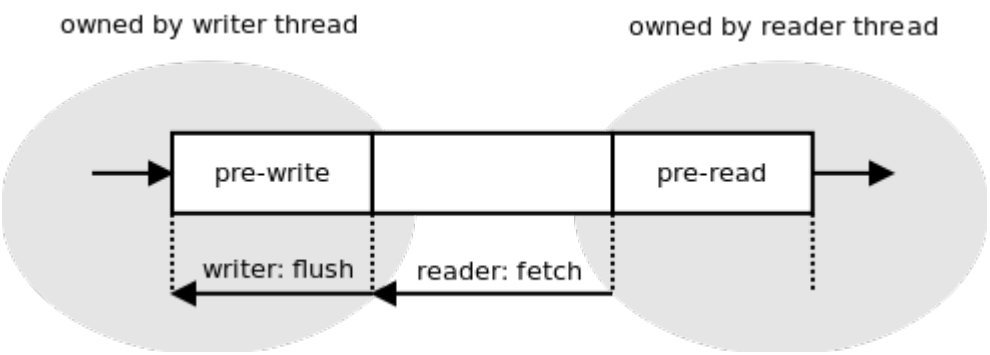


Figure 24.9: Lock-free queue

Lesson learned: Lock-free algorithms are hard to invent, troublesome to implement and almost impossible to debug. If at all possible, use an existing proven algorithm rather than inventing your own. When extreme performance is required, don't rely solely on lock-free algorithms. While they are fast, the performance can be significantly improved by doing smart batching on top of them.

API

The user interface is the most important part of any product. It's the only part of your program visible to the outside world and if you get it wrong the world will hate you. In end-user products it's either the GUI or the command line interface. In libraries it's the API.

In early versions of ØMQ the API was based on AMQP's model of exchanges and queues. (See the AMQP specification.) From a historical perspective it's interesting to have a look at the white paper from 2007 that tries to reconcile AMQP with a brokerless model of messaging. I spent the end of 2009 rewriting it almost from scratch to use the BSD Socket API instead. That was the turning point; ØMQ adoption soared from that point on. While before it was a niche product used by a bunch of messaging experts, afterwards it became a handy commonplace tool for anybody. In a year or so the size of the community increased tenfold, some 20 bindings to different languages were implemented, etc.

The user interface defines the perception of a product. With basically no change to the functionality—just by changing the API—ØMQ changed from an "enterprise messaging" product to a "networking" product. In other words, the perception changed from "a complex piece of infrastructure for big banks" to "hey, this helps me to send my 10-byte-long message from application A to application B".

Lesson learned: Understand what you want your project to be and design the user interface accordingly. Having a user interface that doesn't align with the vision of the project is a 100% guaranteed way to fail.

One of the important aspects of the move to the BSD Sockets API was that it wasn't a revolutionary freshly invented API, but an existing and well-known one. Actually, the BSD Sockets API is one of the oldest APIs still in active use today; it dates back to 1983 and 4.2BSD Unix. It's been widely used and stable for literally decades.

The above fact brings a lot of advantages. Firstly, it's an API that everybody knows, so the learning curve is ludicrously flat. Even if you've never heard of ØMQ, you can build your first application in couple of minutes thanks to the fact that you are able to reuse your BSD Sockets knowledge.

Secondly, using a widely implemented API enables integration of ØMQ with existing technologies. For example, exposing ØMQ objects as "sockets" or "file descriptors" allows for processing TCP, UDP, pipe, file and ØMQ events in the same event loop. Another example: the experimental project to bring ØMQ-like functionality to the Linux kernel turned out to be pretty simple to implement. By sharing the same conceptual framework it can re-use a lot of infrastructure already in place.

Thirdly and probably most importantly, the fact that the BSD Sockets API survived almost three decades despite numerous attempts to replace it means that there is something inherently right in the design. BSD Sockets API designers have—whether deliberately or by chance—made the right design decisions. By adopting the API we can automatically share those design decisions without even knowing what they were and what problem they were solving.

Lesson learned: While code reuse has been promoted from time immemorial and pattern reuse joined in later on, it's important to think of reuse in an even more generic way. When designing a product, have a look at similar products. Check which have failed and which have succeeded; learn from the successful projects. Don't succumb to Not Invented Here syndrome. Reuse the ideas, the APIs, the conceptual frameworks, whatever you find appropriate. By doing so you are allowing users to reuse their existing knowledge. At the same time you may be avoiding technical pitfalls you are not even aware of at the moment.

Messaging Patterns

In any messaging system, the most important design problem is that of how to provide a way for the user to specify which messages are routed to which destinations. There are two main approaches, and I believe this dichotomy is quite generic and applicable to basically any problem encountered in the domain of software.

One approach is to adopt the Unix philosophy of "do one thing and do it well". What this means is that the problem domain should be artificially restricted to a small and well-

understood area. The program should then solve this restricted problem in a correct and exhaustive way. An example of such approach in the messaging area is MQTT. It's a protocol for distributing messages to a set of consumers. It can't be used for anything else (say for RPC) but it is easy to use and does message distribution well.

The other approach is to focus on generality and provide a powerful and highly configurable system. AMQP is an example of such a system. Its model of queues and exchanges provides the user with the means to programmatically define almost any routing algorithm they can think of. The trade-off, of course, is a lot of options to take care of.

ØMQ opts for the former model because it allows the resulting product to be used by basically anyone, while the generic model requires messaging experts to use it. To demonstrate the point, let's have a look how the model affects the complexity of the API. What follows is implementation of RPC client on top of a generic system (AMQP):

```
connect ("192.168.0.111")
exchange.declare (exchange="requests", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
exchange.declare (exchange="replies", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
reply-queue = queue.declare (queue="", passive=false, durable=false,
    exclusive=true, auto-delete=true, no-wait=false, arguments={})
queue.bind (queue=reply-queue, exchange="replies",
    routing-key=reply-queue)
queue.consume (queue=reply-queue, consumer-tag="", no-local=false,
    no-ack=false, exclusive=true, no-wait=true, arguments={})
request = new-message ("Hello World!")
request.reply-to = reply-queue
request.correlation-id = generate-unique-id ()
basic.publish (exchange="requests", routing-key="my-service",
    mandatory=true, immediate=false)
reply = get-message ()
```

On the other hand, ØMQ splits the messaging landscape into so-called "messaging patterns". Examples of the patterns are "publish/subscribe", "request/reply" or "parallelised pipeline". Each messaging pattern is completely orthogonal to other patterns and can be thought of as a separate tool.

What follows is the re-implementation of the above application using ØMQ's request/reply pattern. Note how all the option tweaking is reduced to the single step of choosing the right messaging pattern ("`REQ`"):

```
s = socket (REQ)
s.connect ("tcp://192.168.0.111:5555")
```

```
s.send ("Hello World!")
reply = s.recv ()
```

Up to this point we've argued that specific solutions are better than generic solutions. We want our solution to be as specific as possible. However, at the same time we want to provide our customers with as wide a range of functionality as possible. How can we solve this apparent contradiction?

The answer consists of two steps:

1. Define a layer of the stack to deal with a particular problem area (e.g. transport, routing, presentation, etc.).
2. Provide multiple implementations of the layer. There should be a separate non-intersecting implementation for each use case.

Let's have a look at the example of the transport layer in the Internet stack. It's meant to provide services such as transferring data streams, applying flow control, providing reliability, etc., on the top of the network layer (IP). It does so by defining multiple non-intersecting solutions: TCP for connection-oriented reliable stream transfer, UDP for connectionless unreliable packet transfer, SCTP for transfer of multiple streams, DCCP for unreliable connections and so on.

Note that each implementation is completely orthogonal: a UDP endpoint cannot speak to a TCP endpoint. Neither can a SCTP endpoint speak to a DCCP endpoint. It means that new implementations can be added to the stack at any moment without affecting the existing portions of the stack. Conversely, failed implementations can be forgotten and discarded without compromising the viability of the transport layer as a whole.

The same principle applies to messaging patterns as defined by ØMQ. Messaging patterns form a layer (the so-called "scalability layer") on top of the transport layer (TCP and friends). Individual messaging patterns are implementations of this layer. They are strictly orthogonal—the publish/subscribe endpoint can't speak to the request/reply endpoint, etc. Strict separation between the patterns in turn means that new patterns can be added as needed and that failed experiments with new patterns won't hurt the existing patterns.

Lesson learned: When solving a complex and multi-faceted problem it may turn out that a monolithic general-purpose solution may not be the best way to go. Instead, we can think of the problem area as an abstract layer and provide multiple implementations of this layer, each focused on a specific well-defined use case. When doing so, delineate the use case carefully. Be sure about what is in the scope and what is not. By restricting the use case too aggressively the application of your software may be limited. If you define the problem too

broadly, however, the product may become too complex, blurry and confusing for the users.

Benefits of queue module

As our world becomes populated with lots of small computers connected via the Internet—mobile phones, RFID readers, tablets and laptops, GPS devices, etc.—the problem of distributed computing ceases to be the domain of academic science and becomes a common everyday problem for every developer to tackle. The solutions, unfortunately, are mostly domain-specific hacks. This article summarises our experience with building a large-scale distributed system in a systematic manner. It focuses on problems that are interesting from a software architecture point of view, and we hope that designers and programmers in the open source community will find it useful.

Application engine

These days the Internet is so widespread and ubiquitous it’s hard to imagine it wasn’t exactly there, as we know it, a decade ago. With the proliferation of permanently connected PCs, mobile devices and recently tablets, the Internet landscape is rapidly changing and entire economies have become digitally wired. Online services have become much more elaborate with a clear bias towards instantly available live information and entertainment. Security aspects of running online business have also significantly changed. Accordingly, websites are now much more complex than before, and generally require a lot more engineering efforts to be robust and scalable.

One of the biggest challenges for a website architect has always been concurrency. Since the beginning of web services, the level of concurrency has been continuously growing. It’s not uncommon for a popular website to serve hundreds of thousands and even millions of simultaneous users.

Nowadays, concurrency is caused by a combination of mobile clients and newer application architectures which are typically based on maintaining a persistent connection that allows the client to be updated with news, tweets, friend feeds, and so on. Another important factor contributing to increased concurrency is the changed behavior of modern browsers, which open four to six simultaneous connections to a website to improve page load speed.

With persistent connections the problem of handling concurrency is even more pronounced, because to avoid latency associated with establishing new HTTP connections, clients would stay connected, and for each connected client there’s a certain amount of memory allocated by the web server.

Advantages of using this architecture

Handling high concurrency with high performance and efficiency, there are now even more interesting benefits.

With recent flavors of development kits and programming languages gaining wide use, more and more companies are changing their application development and deployment habits.

Overall Architecture

Traditional process- or thread-based models of handling concurrent connections involve handling each connection with a separate process or thread, and blocking on network or input/output operations. Depending on the application, it can be very inefficient in terms of memory and CPU consumption. Spawning a separate process or thread requires preparation

of a new runtime environment, including allocation of heap and stack memory, and the creation of a new execution context.

Additional CPU time is also spent creating these items, which can eventually lead to poor performance due to thread thrashing on excessive context switching. All of these complications manifest themselves in older web server architectures like Apache's. This is a tradeoff between offering a rich set of generally applicable features and optimized usage of server resources.

The app engine uses multiplexing and event notifications heavily, and dedicates specific tasks to separate processes. Connections are processed in a highly efficient run-loop in a limited number of single-threaded processes called `worker`s. Within each `worker` nginx can handle many thousands of concurrent connections and requests per second.

Code Structure

The `worker` code includes the core and the functional modules. The core of the module is responsible for maintaining a tight run-loop and executing appropriate sections of modules' code on each stage of request processing. Modules constitute most of the presentation and application layer functionality. Modules read from and write to the network and storage, transform content, do outbound filtering, apply server-side include actions and pass the requests to the upstream servers when proxying is activated.

The app engine's modular architecture generally allows developers to extend the set of web server features without modifying the core. Modules come in slightly different incarnations, namely core modules, event modules, phase handlers, protocols, variable handlers, filters, upstreams and load balancers. At this time, it doesn't support dynamically loaded modules; i.e., modules are compiled along with the core at build stage. However, support for loadable modules and ABI is planned for the future major releases.

While handling a variety of actions associated with accepting, processing and managing network connections and content retrieval, uses event notification mechanisms and a number of disk I/O performance enhancements in Linux, Solaris and BSD-based operating systems, like `kqueue`, `epoll`, and `event ports`.

The goal is to provide as many hints to the operating system as possible, in regards to obtaining timely asynchronous feedback for inbound and outbound traffic, disk operations, reading from or writing to sockets, timeouts and so on. The usage of different methods for multiplexing and advanced I/O operations is heavily optimized for every Unix-based operating systems

A high-level overview of architecture is presented below

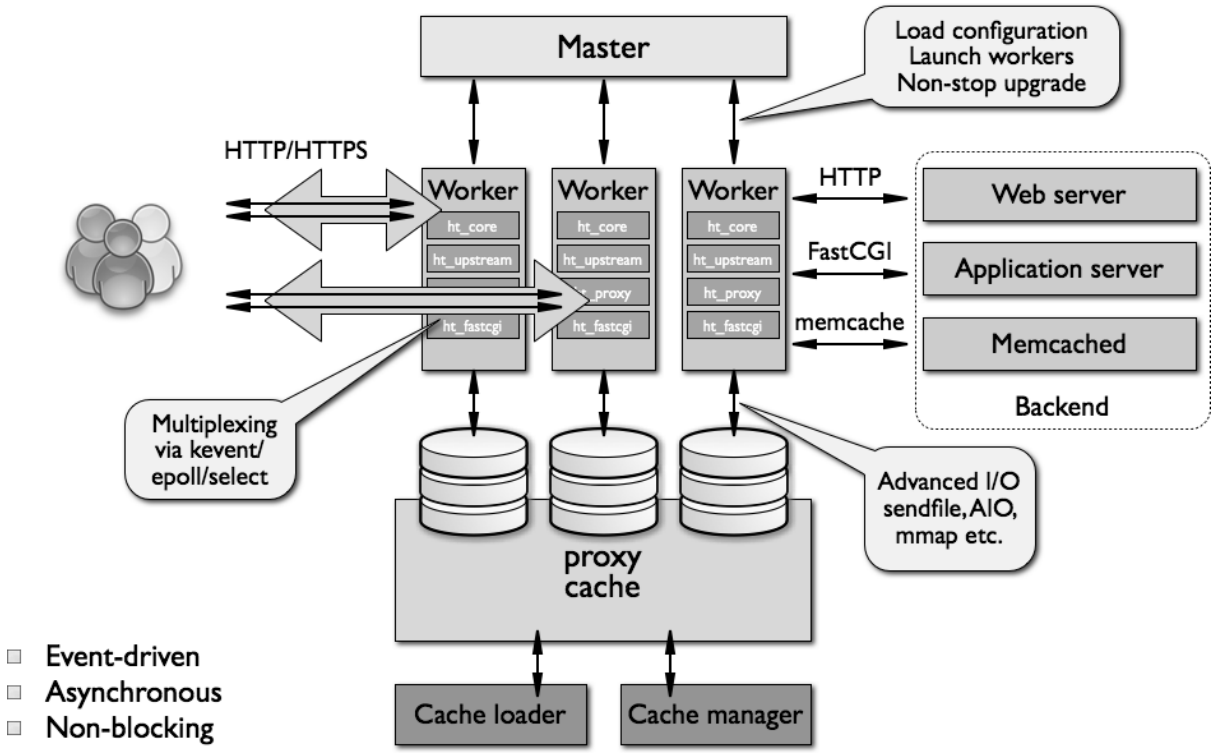


Figure 14.1: Diagram of app engine architecture

Workers Model

As previously mentioned, the engine doesn't spawn a process or thread for every connection. Instead, `worker` processes accept new requests from a shared "listen" socket and execute a highly efficient run-loop inside each `worker` to process thousands of connections per `worker`.

There's no specialized arbitration or distribution of connections to the `worker`s in nginx; this work is done by the OS kernel mechanisms. Upon startup, an initial set of listening sockets is created. `worker`s then continuously accept, read from and write to the sockets while processing HTTP requests and responses.

The run-loop is the most complicated part of the `worker` code. It includes comprehensive inner calls and relies heavily on the idea of asynchronous task handling. Asynchronous operations are implemented through modularity, event notifications, extensive use of callback functions and fine-tuned timers. Overall, the key principle is to be as non-blocking as possible. The only situation where engine can still block is when there's not enough disk storage performance for a `worker` process.

Because it does not fork a process or thread per connection, memory usage is very conservative and extremely efficient in the vast majority of cases. It conserves CPU cycles as well because there's no ongoing create-destroy pattern for processes or threads.

What the engine does is check the state of the network and storage, initialize new

connections, add them to the run-loop, and process asynchronously until completion, at which point the connection is deallocated and removed from the run-loop. Combined with the careful use of `syscall`s and an accurate implementation of supporting interfaces like pool and slab memory allocators, it typically achieves moderate-to-low CPU usage even under extreme workloads.

Because the engine spawns several `worker`s to handle connections, it scales well across multiple cores. Generally, a separate `worker` per core allows full utilization of multicore architectures, and prevents thread thrashing and lock-ups. There's no resource starvation and the resource controlling mechanisms are isolated within single-threaded `worker` processes. This model also allows more scalability across physical storage devices, facilitates more disk utilization and avoids blocking on disk I/O. As a result, server resources are utilized more efficiently with the workload shared across several workers.

Another problem with the existing `worker` model is related to limited support for embedded scripting. For one, with the standard distribution, only embedding Perl scripts is supported. There is a simple explanation for that: the key problem is the possibility of an embedded script to block on any operation or exit unexpectedly. Both types of behavior would immediately lead to a situation where the worker is hung, affecting many thousands of connections at once. More work is planned to make embedded scripting for a simpler, more reliable and suitable for a broader range of applications.

Process Roles

There are several processes in memory; there is a single master process and several `worker` processes. There are also a couple of special purpose processes, specifically a cache loader and cache manager. All processes are single-threaded in version 1.x. All processes primarily use shared-memory mechanisms for inter-process communication. The master process is run as the `root` user. The cache loader, cache manager and `worker`s run as an unprivileged user.

The master process is responsible for the following tasks:

- reading and validating configuration
- creating, binding and closing sockets
- starting, terminating and maintaining the configured number of `worker` processes
- reconfiguring without service interruption
- controlling non-stop binary upgrades (starting new binary and rolling back if necessary)
- re-opening log files

- compiling embedded Perl scripts

The `worker` processes accept, handle and process connections from clients, provide reverse proxying and filtering functionality and do almost everything else is capable of. In regards to monitoring the behavior of an instance, a system administrator should keep an eye on `worker`s as they are the processes reflecting the actual day-to-day operations of a web server.

The cache loader process is responsible for checking the on-disk cache items and populating the in-memory database with cache metadata. Essentially, the cache loader prepares instances to work with files already stored on disk in a specially allocated directory structure. It traverses the directories, checks cache content metadata, updates the relevant entries in shared memory and then exits when everything is clean and ready for use.

The cache manager is mostly responsible for cache expiration and invalidation. It stays in memory during normal operation and it is restarted by the master process in the case of failure.

Brief Overview of Caching

Caching is implemented in the form of hierarchical data storage on a filesystem. Cache keys are configurable, and different request-specific parameters can be used to control what gets into the cache. Cache keys and cache metadata are stored in the shared memory segments, which the cache loader, cache manager and `worker`s can access. Currently there is not any in-memory caching of files, other than optimizations implied by the operating system’s virtual filesystem mechanisms. Each cached response is placed in a different file on the filesystem. The hierarchy (levels and naming details) are controlled through configuration directives. When a response is written to the cache directory structure, the path and the name of the file are derived from an MD5 hash of the proxy URL.

The process for placing content in the cache is as follows: When reads the response from an upstream server, the content is first written to a temporary file outside of the cache directory structure. When finishes processing the request it renames the temporary file and moves it to the cache directory. If the temporary files directory for proxying is on another file system, the file will be copied, thus it’s recommended to keep both temporary and cache directories on the same file system. It is also quite safe to delete files from the cache directory structure when they need to be explicitly purged. There are third-party extensions for which make it possible to control cached content remotely, and more work is planned to integrate this functionality in the main distribution.

Configuration

Configuration system was inspired by Igor Sysoev's experiences with Apache. His main insight was that a scalable configuration system is essential for a web server. The main scaling problem was encountered when maintaining large complicated configurations with lots of virtual servers, directories, locations and datasets. In a relatively big web setup it can be a nightmare if not done properly both at the application level and by the system engineer himself.

As a result, configuration was designed to simplify day-to-day operations and to provide an easy means for further expansion of web server configuration.

Internals

As was mentioned before, the codebase consists of a core and a number of modules. The core of is responsible for providing the foundation of the web server, web and mail reverse proxy functionalities; it enables the use of underlying network protocols, builds the necessary run-time environment, and ensures seamless interaction between different modules. However, most of the protocol- and application-specific features are done by modules, not the core.

Internally, processes connections through a pipeline, or chain, of modules. In other words, for every operation there's a module which is doing the relevant work; e.g., compression, modifying content, executing server-side includes, communicating to the upstream application servers through FastCGI or uwsgi protocols, or talking to memcached.

There are a couple of modules that sit somewhere between the core and the real "functional" modules. These modules are `http` and `mail`. These two modules provide an additional level of abstraction between the core and lower-level components. In these modules, the handling of the sequence of events associated with a respective application layer protocol like HTTP, SMTP or IMAP is implemented. In combination with the core, these upper-level modules are responsible for maintaining the right order of calls to the respective functional modules. While the HTTP protocol is currently implemented as part of the `http` module, there are plans to separate it into a functional module in the future, due to the need to support other protocols like SPDY;.

The functional modules can be divided into event modules, phase handlers, output filters, variable handlers, protocols, upstreams and load balancers. Most of these modules complement the HTTP functionality, though event modules and protocols are also used for `mail`. Event modules provide a particular OS-dependent event notification mechanism like `kqueue` or `epoll`. The event module that uses depends on the operating system capabilities and build configuration. Protocol modules allow to communicate through HTTPS, TLS/SSL,

SMTP, POP3 and IMAP.

A typical HTTP request processing cycle looks like the following.

1. Client sends HTTP request.
2. the core chooses the appropriate phase handler based on the configured location matching the request.
3. If configured to do so, a load balancer picks an upstream server for proxying.
4. Phase handler does its job and passes each output buffer to the first filter.
5. First filter passes the output to the second filter.
6. Second filter passes the output to third (and so on).
7. Final response is sent to the client.

Module invocation is extremely customizable. It is performed through a series of callbacks using pointers to the executable functions. However, the downside of this is that it may place a big burden on programmers who would like to write their own modules, because they must define exactly how and when the module should run. Both the API and developers' documentation are being improved and made more available to alleviate this.

Some examples of where a module can attach are:

- Before the configuration file is read and processed
- For each configuration directive for the location and the server where it appears
- When the main configuration is initialized
- When the server (i.e., host/port) is initialized
- When the server configuration is merged with the main configuration
- When the location configuration is initialized or merged with its parent server configuration
- When the master process starts or exits
- When a new worker process starts or exits
- When handling a request
- When filtering the response header and the body
- When picking, initiating and re-initiating a request to an upstream server
- When processing the response from an upstream server
- When finishing an interaction with an upstream server

Inside a `worker`, the sequence of actions leading to the run-loop where the response is generated looks like the following:

1. Begin `ngx_worker_process_cycle()`.
2. Process events with OS specific mechanisms (such as `epoll` or `kqueue`).
3. Accept events and dispatch the relevant actions.
4. Process/proxy request header and body.
5. Generate response content (header, body) and stream it to the client.
6. Finalize request.
7. Re-initialize timers and events.

The run-loop itself (steps 5 and 6) ensures incremental generation of a response and streaming it to the client.

A more detailed view of processing an HTTP request might look like this:

1. Initialize request processing.
2. Process header.
3. Process body.
4. Call the associated handler.
5. Run through the processing phases.

Which brings us to the phases. When handles an HTTP request, it passes it through a number of processing phases. At each phase there are handlers to call. In general, phase handlers process a request and produce the relevant output. Phase handlers are attached to the locations defined in the configuration file.

Phase handlers typically do four things: get the location configuration, generate an appropriate response, send the header, and send the body. A handler has one argument: a specific structure describing the request. A request structure has a lot of useful information about the client request, such as the request method, URI, and header.

When the HTTP request header is read, the engine does a lookup of the associated virtual server configuration. If the virtual server is found, the request goes through six phases:

1. server rewrite phase
2. location phase

3. location rewrite phase (which can bring the request back to the previous phase)
4. access control phase
5. try_files phase
6. log phase

In an attempt to generate the necessary content in response to the request, passes the request to a suitable content handler. Depending on the exact location configuration, it may try so-called unconditional handlers first, like `perl`, `proxy_pass`, `flv`, `mp4`, etc. If the request does not match any of the above content handlers, it is picked by one of the following handlers, in this exact order: `random index`, `index`, `autoindex`, `gzip_static`, `static`.

The content handlers' content is then passed to the filters. Filters are also attached to locations, and there can be several filters configured for a location. Filters do the task of manipulating the output produced by a handler. The order of filter execution is determined at compile time. For the out-of-the-box filters it's predefined, and for a third-party filter it can be configured at the build stage. In the existing implementation, filters can only do outbound changes and there is currently no mechanism to write and attach filters to do input content transformation. Input filtering will appear in future versions.

Filters follow a particular design pattern. A filter gets called, starts working, and calls the next filter until the final filter in the chain is called. After that, it finalizes the response. Filters don't have to wait for the previous filter to finish. The next filter in a chain can start its own work as soon as the input from the previous one is available (functionally much like the Unix pipeline). In turn, the output response being generated can be passed to the client before the entire response from the upstream server is received.

There are header filters and body filters;It feeds the header and the body of the response to the associated filters separately.

A header filter consists of three basic steps:

1. Decide whether to operate on this response.
2. Operate on the response.
3. Call the next filter.

Body filters transform the generated content. Examples of body filters include:

- server-side includes
- XSLT filtering

- image filtering (for instance, resizing images on the fly)
- charset modification
- `gzip` compression
- chunked encoding

After the filter chain, the response is passed to the writer. Along with the writer there are a couple of additional special purpose filters, namely the `copy` filter, and the `postpone` filter. The `copy` filter is responsible for filling memory buffers with the relevant response content which might be stored in a proxy temporary directory. The `postpone` filter is used for subrequests.

Upstream and load balancers are also worth describing briefly.

Upstreams are used to implement what can be identified as a content handler which is a reverse proxy (`proxy_pass` handler). Upstream modules mostly prepare the request to be sent to an upstream server (or "backend") and receive the response from the upstream server. There are no calls to output filters here. What an upstream module does exactly is set callbacks to be invoked when the upstream server is ready to be written to and read from. Callbacks implementing the following functionality exist:

- Crafting a request buffer (or a chain of them) to be sent to the upstream server
- Re-initializing/resetting the connection to the upstream server (which happens right before creating the request again)
- Processing the first bits of an upstream response and saving pointers to the payload received from the upstream server
- Aborting requests (which happens when the client terminates prematurely)
- Finalizing the request when finishes reading from the upstream server
- Trimming the response body (e.g. removing a trailer)

Load balancer modules attach to the `proxy_pass` handler to provide the ability to choose an upstream server when more than one upstream server is eligible. A load balancer registers an enabling configuration file directive, provides additional upstream initialization functions (to resolve upstream names in DNS, etc.), initializes the connection structures, decides where to route the requests, and updates stats information. Currently supports two standard disciplines for load balancing to upstream servers: round-robin and ip-hash.

Upstream and load balancing handling mechanisms include algorithms to detect failed upstream servers and to re-route new requests to the remaining ones—though a lot of additional work is planned to enhance this functionality. In general, more work on load

balancers is planned, and in the next versions of the mechanisms for distributing the load across different upstream servers as well as health checks will be greatly improved.

There are also a couple of other interesting modules which provide an additional set of variables for use in the configuration file. While the variables in are created and updated across different modules, there are two modules that are entirely dedicated to variables: `geo` and `map`. The `geo` module is used to facilitate tracking of clients based on their IP addresses. This module can create arbitrary variables that depend on the client's IP address. The other module, `map`, allows for the creation of variables from other variables, essentially providing the ability to do flexible mappings of hostnames and other run-time variables. This kind of module may be called the variable handler.

Memory allocation mechanisms implemented inside a single `worker` were, to some extent, inspired by Apache. A high-level description of memory management would be the following: For each connection, the necessary memory buffers are dynamically allocated, linked, used for storing and manipulating the header and body of the request and the response, and then freed upon connection release.

Going a bit deeper, when the response is generated by a module, the retrieved content is put in a memory buffer which is then added to a buffer chain link. Subsequent processing works with this buffer chain link as well. Buffer chains are quite complicated because there are several processing scenarios which differ depending on the module type. For instance, it can be quite tricky to manage the buffers precisely while implementing a body filter module. Such a module can only operate on one buffer (chain link) at a time and it must decide whether to overwrite the input buffer, replace the buffer with a newly allocated buffer, or insert a new buffer before or after the buffer in question. To complicate things, sometimes a module will receive several buffers so that it has an incomplete buffer chain that it must operate on.

A note on the above approach is that there are memory buffers allocated for the entire life of a connection, thus for long-lived connections some extra memory is kept..