دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

کارشناسی ارشد
مهندسی نرم‌افزار

عنوان

الگوها در مهندسی نرم‌افزار

تمرین دوم

نگارش

مهدی عیدی

استاد

دکتر رامان رامسین

بهار ۱۴۰۴

چکیده

در این تمرین، در بخش نخست، سیستم هدف مورد بررسی قرار گرفته، نیازمندی‌ها استخراج شده و با ترکیب الگوهای طراحی و معماری یک راه‌حل برای آن پیشنهاد شده و سپس راه‌حل پیشنهاد شده ارزیابی شده است. در بخش دوم دو Idiom در زبان جاوا معرفی شدند.

کلیدواژه‌ها: الگوهای طراحی، الگوهای معماری، طراحی معماری، ترکیب الگوها، Idiom

فهرست مطالب

فهرست تصاویر

فصل ۱

طراحی معماری

در این قسمت، سیستم توصیف شده در تمرین مورد بررسی قرار گرفته، نیازمندی‌ها استخراج شده، با ترکیب الگوهای GoF و GoV یک معماری برای سیستم هدف پیشنهاد شده و راه‌حل پیشنهاد شده ارزیابی می‌شود. [۲] [۳]

۱-۱ شرح مسئله

در یک سیستم توزیع شده، زیرسیستم‌ها با یکدیگر تعامل دارند تا خدماتی را به مشتریان سیستم ارائه دهند. الگوریتم تعامل بسته به نوع خدمت[۱] درخواستی و محدودیت‌های فعلی سیستم در منابع موجود (از جمله منابع ارتباطی، پردازشی و ذخیره‌سازی) متفاوت است. مشتریان دارای رتبه‌بندی هستند و تخصیص منابع بیشتر برای خدمت‌رسانی به مشتریان با رتبه بالاتر مجاز است. هدف این است که سامانه به‌گونه‌ای طراحی شود که الگوریتم تعامل بتواند در زمان اجرا بر اساس پارامترهای مذکور تطبیق یابد.

۱-۱-۱ ویژگی‌های مسئله

- با یک سیستم توزیع شده سروکار داریم. زیرسیستم‌های متعدد که می‌توانند روی ماشین‌های مختلف اجرا شوند، با یکدیگر کار می‌کنند.
- زیرسیستم‌ها با یکدیگر تعامل می‌کنند. این تعامل می‌تواند با روش‌هایی مانند ارسال پیغام[۲]، فراخوانی

[۱] service
[۲] message passing

رویه از راه دور یا RPC[۳] یا از طریق API[۴] باشد.

- زیرسیستم‌ها خدماتی را به مشتریان ارائه می‌کنند. مشتریان می‌توانند کاربران خارجی یا سایر سیستم‌ها باشند که مصرف‌کننده و استفاده کننده از وظیفه‌مندی هستند.
- الگوریتم تعامل زیرسیستم‌ها ایستا نیست و در زمان اجرا تغییر کرده و تطبیق پیدا می‌کند.
- عوامل تاثیرگذار روی الگوریتم تعامل شامل موارد زیر است:
  - سرویس درخواست شده.
  - وضعیت فعلی منابع سیستم مانند منابع ارتباطی مثل پهنای باند، منابع پردازشی مثل CPU و منابع ذخیره‌سازی مثل مموری.
- یک سیستم اولویت‌بندی وجود دارد و مشتری‌هایی که رتبه بالاتری دارند، مجاز هستند برای دریافت سرویس درخواستی، از منابع بیشتری استفاده کنند.
- سیستم باید از رتبه‌بندی مشتریان آگاه بوده و مطابق آن رفتار کند. مثلا الگوریتمی متفاوت اجرا کند که سریع‌تر پاسخ می‌دهد ولی منابع بیشتری مصرف می‌کند.

۱-۱-۲ هدف

هدف این است که یک سیستم توزیع شده طراحی کنیم که در آن زیرسیستم‌ها به صورت منعطف و قابل تطبیق با یکدیگر تعامل می‌کنند و الگوریتم تعامل می‌تواند در زمان اجرا به‌صورت پویا عوض شود. تغییر و تطبیق الگوریتم نیز وابسته به سرویس درخواستی، منابع کنونی سیستم و رتبه‌بندی مشتری است.

۱-۱-۳ نیازمندی‌ها و اجبارها

- توزیع‌شدگی مولفه‌های سیستم و تعامل توزیع شده مولفه‌ها.
- تطبیق و تغییر الگوریتم تعامل در زمان اجرا به‌صورت پویا.
- طراحی ماجولار[۵]. زیرسیستم‌ها باید loosely coupled باشند تا انعطاف‌پذیری در تعامل آن‌ها وجود داشته باشد.
- پایش و آگاهی از وضعیت کنونی منابع سیستم.

---

[۳] Remote Procedure Call
[۴] Application Programming Interface
[۵] modular

- آگاهی از رتبه‌بندی مشتریان و اعمال سیاست‌های مناسب مطابق آن.

۱-۲ راه‌حل

در این قسمت برای زیرمسائل مطرح شده در قسمت قبل، از میان الگوهای طراحی و معماری، موارد مناسب همراه با توضیحات آورده شده است.

۱-۲-۱ توزیع‌شدگی

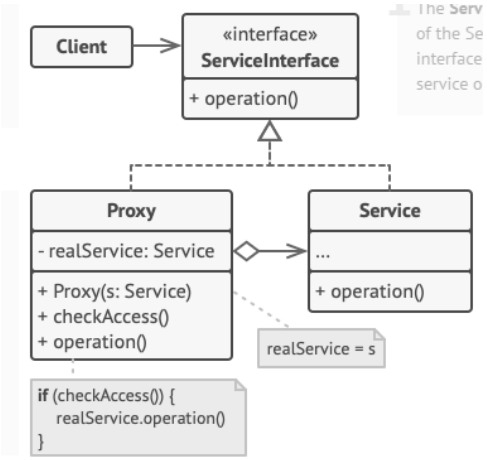
الگوی پیشنهادی: Broker

همانطور که بیان شد، مولفه‌های سیستم به‌صورت توزیع شده بوده و با یکدیگر تعامل می‌کنند. به‌منظور ارائه بستری برای توزیع شدگی و تعامل توزیع شده مولفه‌ها از الگوی Broker می‌توان استفاده کرد. این الگو این امکان را می‌دهد تا سرورها، سرویس‌های خود را در بروکر ثبت کنند و مشتریان سیستم با رجوع به بروکر، بدون نیاز به داشتن آدرس فیزیکی سرورها، می‌توانند سرویس‌های موردنیاز خود را فراخوانی کنند و جزئیات سطح پایین پیدا کردن آدرس فیزیکی و ارتباطات شبکه‌ای بر دوش بروکر خواهد بود. همچنین اگر سیستم‌های توزیع شده نامتجانس هم وجود داشته باشد، این الگو از طریق Bridgeها این امکان را می‌دهد تا تعامل در سطح سیستم‌های توزیع شده نامتجانس هم با وابستگی و جفت‌شدگی[۶] پایین انجام شود.

الگوی پیشنهادی: Remote Proxy

این الگو در دل الگوی Broker وجود دارد. در سمت مشتری، یک Client-Side Proxy وجود دارد که این توهم را به مشتری می‌دهد که گویا سرویس فراخوانی شده به صورت محلی وجود دارد و نه توزیع شده و در سروری جدا. این پروکسی، دغدغه ارتباط مشتری با بروکر را از دوش مشتری بر می‌دارد. مشتری فکر می‌کند که با یک شی محلی ارتباط گرفته و سرویس آن را فراخوانی می‌کند. ولی درواقع پروکسی سمت مشتری این درخواست را گرفته، با بروکر ارتباط برقرار کرده و به آن ارسال می‌کند و در نهایت پاسخ را دریافت و به مشتری می‌دهد. همچنین در سمت سرورها نیز Server-Side Proxy وجود دارد که این توهم را به سرور می‌دهد که گویا فراخوانی کننده سرویس آن‌ها یک شی محلی است و آن‌ها پاسخ را به آن شی محلی بازمی‌گردانند. در حالی که این پروکسی سمت سرور سرویس مربوطه را از سرور فراخوانی کرده، پاسخ را دریافت و به بروکر ارسال می‌کند و دغدغه سطح پایین ارتباط با بروکر بر دوش پروکسی است. (شکل ۱-۱)

[۶] coupling

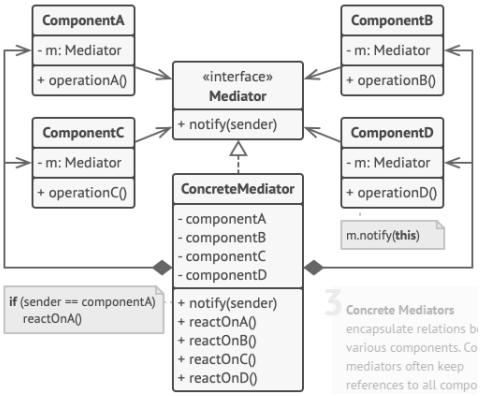


شکل ۱-۱: ساختار الگوی پروکسی [۱]

۱-۲-۲ تغییر پویای الگوریتم تعامل

الگوی پیشنهادی: Mediator

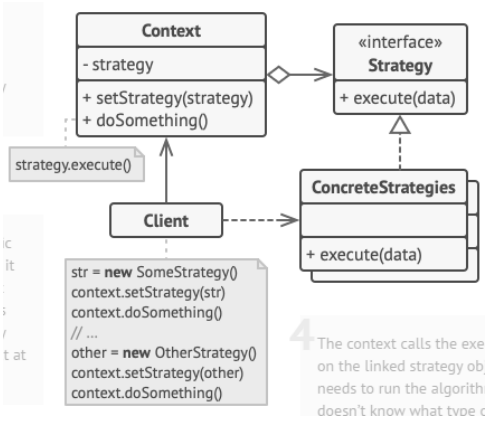
چون زیرسیستم‌ها با یکدیگر تعامل می‌کنند، اگر یک شی مرکزی به عنوان کنترل کننده تعامل وجود نداشته باشد، الگوریتم تعامل بین زیرسیستم‌ها توزیع شده می‌شود. این موضوع، درک، تغییر و تطبیق الگوریتم را سخت می‌کند. بنابراین شی مرکزی نیاز است. این شی مرکزی که اجرا کننده الگوریتم تعامل است، داخل مایکروکرنل در نظر گرفته شده است که جلوتر توضیح داده می‌شود. اجرای فیزیکی الگوریتم از طریق Broker انجام می‌شود و تصمیم و مشخص کردن استراتژی الگوریتم در مولفه Microkernel انجام می‌شود. (شکل ۱-۲)



شکل ۱-۲: ساختار الگوی Mediator [۱]

الگوی پیشنهادی: Strategy

الگوریتم‌های تعامل مختلف در قالب اشیای استراتژی قرار دارند. در زمان اجرا، مولفه تصمیم‌گیر که بعدا توضیح داده خواهد شد، براساس معیارهای بیان شده، شی استراتژی مناسب که حاوی الگوریتم تعامل است را ساخته و مولفه اجرا کننده را با آن پیکربندی می‌کند. (شکل ۱-۳)

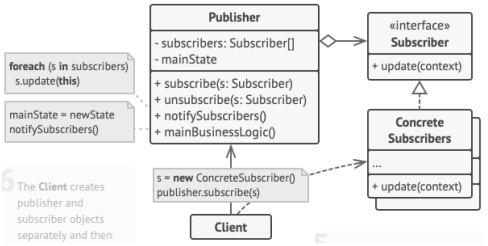


شکل ۱-۳: ساختار الگوی Strategy [۱]

۱-۲-۳ پایش منابع سیستم

الگوی پیشنهادی: Observer

در سیستم یک مولفه که کار پایش منابع سیستم را انجام می‌دهد وجود دارد و نقش سابجکت را بازی می‌کند. مولفه تصمیم‌گیر استراتژی، نقش Observer برای آن را بازی می‌کند. زمانی که تغییرات عمده‌ای در منابع سیستم رخ داد، برای مثال پر شدن ممووری، پیغام آپدیت به مولفه تصمیم‌گیر ارسال می‌شود. مولفه تصمیم‌گیر نیز اطلاعات لازم را دریافت کرده و حالت درونی خود را تنظیم می‌کند. این اطلاعات برای تصمیم در رابطه با الگوریتم بکار گرفته خواهند شد. (شکل ۱-۴)



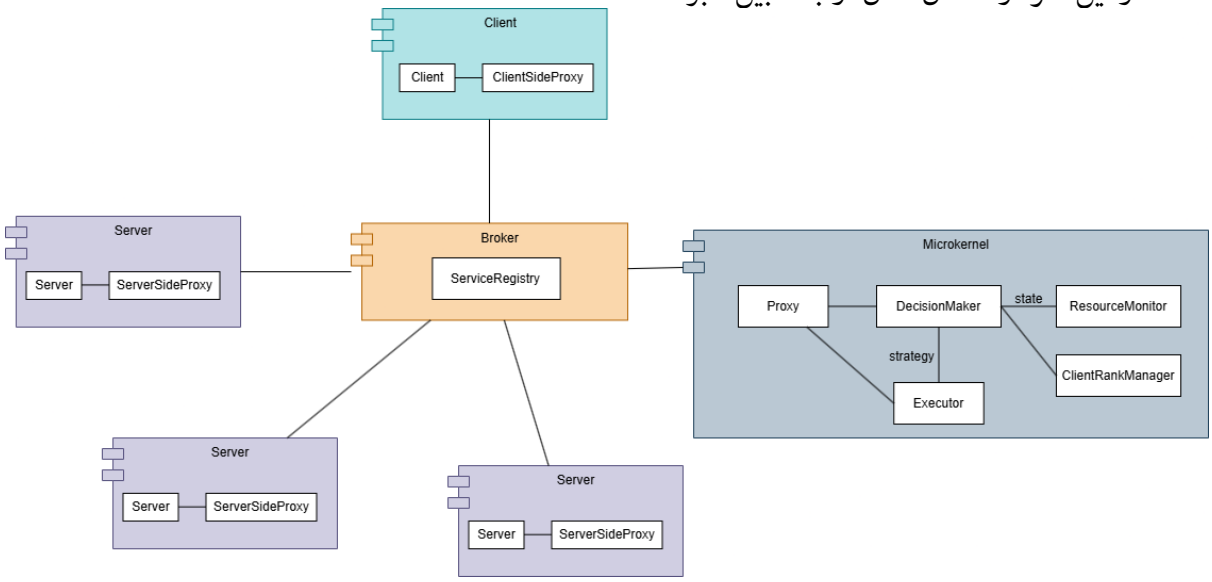
شکل ۱-۴: ساختار الگوی Observer [۱]

الگوی پیشنهادی: Microkernel

در کل مواردی چون پایش منابع سیستم، تصمیم‌گیری استراتژی و الگوریتم تعامل و مدیریت رتبه‌بندی مشتری‌ها در یک مولفه مایکروکرنل متمرکز شده‌اند و این سرویس‌ها در اختیار مشتری بیرونی که عمدتا بروکر هست قرار می‌گیرند.

۱-۲-۴ نمودار مفهومی از معماری کلی

در شکل ۱-۵ نمودار ساده مفهومی از معماری پیشنهادی مشاهده می‌شود (در این نمودار حالت سیستم‌های توزیع شده نامتجانس با Bridgeها به منظور سادگی نمایش داده نشده است. در ادامه به آن نیز پرداخته می‌شود). هدف از این نمودار، نشان دادن ارتباط بین اجزا است.



شکل ۱-۵: نمودار مفهومی معماری پیشنهادی

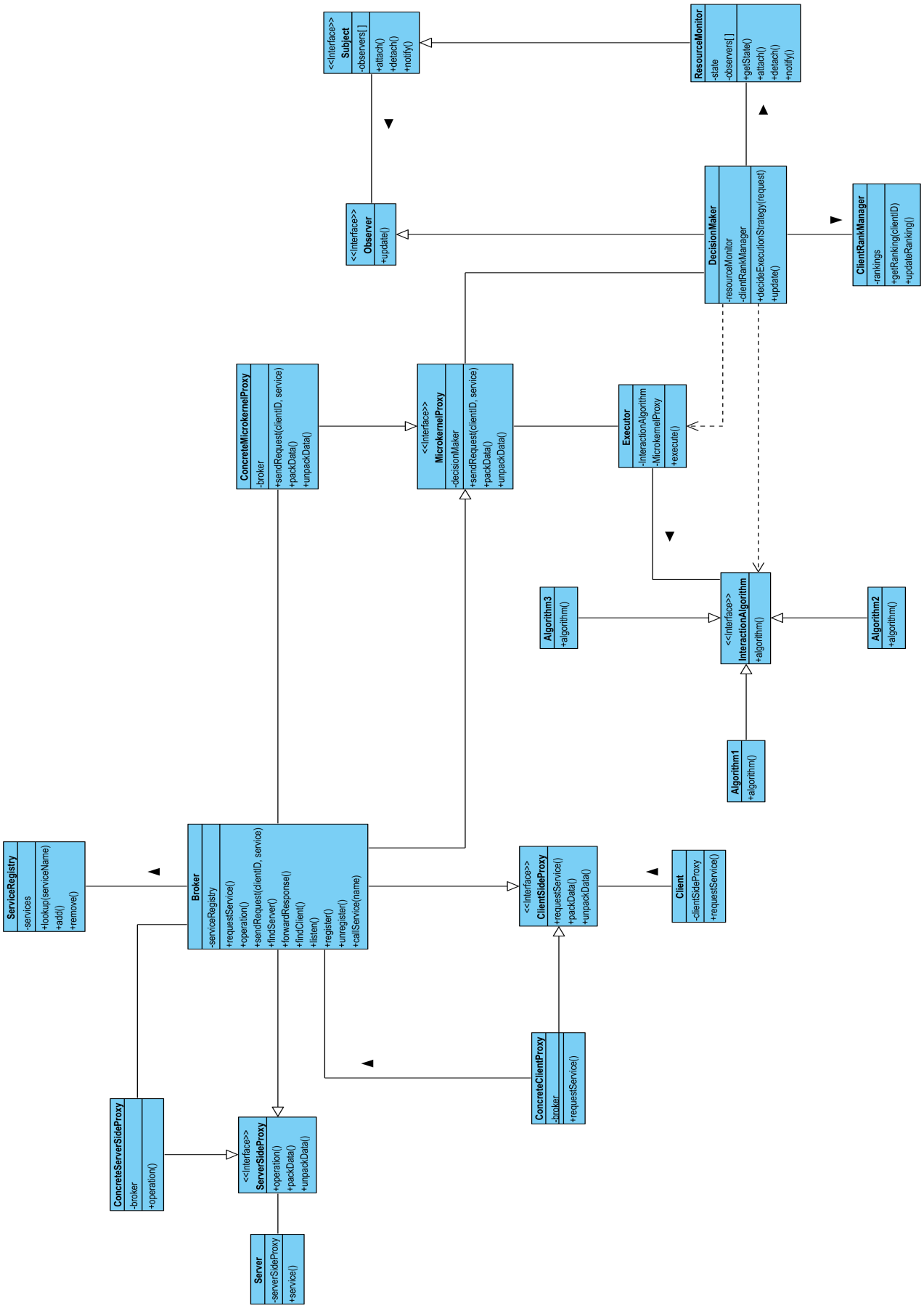
در معماری پیشنهادی، مولفه‌های مشتری، با پروکسی‌های سمت مشتری ارتباط دارند. مشتری‌ها هیچ دغدغه‌ای مبنی بر ارتباط با بروکر، یافتن سرویس مورد نظر و آدرس فیزیکی، ارتباطات سطح پایین و شبکه‌ای و دغدغه‌های اینچنینی ندارند. اساسا این پروکسی توهم محلی بودن سرویس را به مشتری می‌دهد. پروکسی سمت مشتری درخواست را از مشتری دریافت کرده و با بروکر ارتباط برقرار می‌کند و درخواست سرویس را به آن ارسال می‌کند. بروکر پس از دریافت درخواست از سمت پروکسی مشتری، برای اجرای الگوریتم تعامل بین زیرسیستم‌ها نیاز دارد با مولفه مایکروکرنل ارتباط برقرار کند. بدین منظور درخواستی را به مایکروکرنل ارسال کرده و در آن سرویس درخواستی و شناسه کاربر را ارسال می‌کند. مولفه مایکروکرنل نیز از طریق پروکسی با بروکر ارتباط دارد. داخل مولفه مایکروکرنل، درخواست وارد مولفه تصمیم‌گیر می‌شود. این مولفه یک Observer برای مولفه پایش منابع سیستم است و وضعیتش متناسب با آن بروز می‌شود. همچنین با مولفه رتبه‌بندی مشتری‌ها نیز ارتباط داشته و اطلاعات لازم در رابطه با رتبه‌بندی مشتری را از طریق شناسه آن دریافت می‌کند. سپس با تجمیع این اطلاعات، تصمیم مربوط به الگوریتم تعامل را گرفته و شی استراتژی مربوطه که الگوریتم تعامل را پیاده‌سازی می‌کند، ساخته و مولفه اجرا کننده را با آن پیکربندی می‌کند. مولفه اجرا کننده داخل مایکروکرنل پس از دریافت شی استراتژی که حاوی الگوریتم تعامل است، از طریق پروکسی با بروکر ارتباط گرفته، الگوریتم تعامل و کارچرخانی را انجام داده و سرویس‌های لازم را از طریق بروکر فراخوانی می‌کند. سپس پاسخ نهایی را به بروکر بازمیگرداند. سرورها نیز سرویس‌های خود را در بروکر ثبت می‌کنند و همچنین برای ارتباط با

بروکر نیز از طریق پروکسی سمت خود عمل می‌کنند. بدین شکل در زمان اجرا، بر اساس رتبه‌بندی مشتری، وضعیت کنونی منابع سیستم و سرویس درخواستی، یک الگوریتم تعامل انتخاب شده و اجرا می‌شود که این انعطاف‌پذیری لازم را به ما می‌دهد.

۱-۲-۵ ساختار راه‌حل

در شکل ۱-۶ نمودار کلاس سطح بالای راه‌حل پیشنهادی نمایش داده شده است.

همان‌طور که در نمودار کلاس دیده می‌شود، مشتری به واسط پروکسی دید داشته و با آن کار می‌کند. پروکسی سمت مشتری نیز یک ریموت پروکسی بوده و با بروکر ارتباط می‌گیرد. موضوع مشابه برای سرورها نیز وجود دارد. بروکر نیز برای ترجمه نام سرویس‌ها به آدرس فیزیکی با یک ServiceRegistry ارتباط داشته و نگاشت از طریق آن صورت می‌پذیرد. درخواست‌هایی که از سمت مشتری وارد بروکر می‌شوند، برای اجرای الگوریتم تعامل بین سرویس‌ها و پاسخ دادن به درخواست، نیاز دارد تا درخواست را به مولفه مایکروکرنل ارسال کند. بدین منظور به پروکسی سمت مایکروکرنل ارسال کرده و پروکسی نیز درخواست را به مولفه DecisionMaker ارسال می‌کند. این مولفه یک Observer است برای مولفه پایش منابع سیستم تا از وضعیت فعلی منابع سیستم مطلع باشد. همچنین با مولفه رتبه‌بندی مشتری نیز ارتباط دارد تا اطلاعات لازم درباره رتبه مشتری را دریافت کند. با داشتن این اطلاعات، می‌تواند در زمان اجرا تصمیم بگیرد که از چه الگوریتمی برای تعامل بین سرویس‌ها و پاسخ به مشتری استفاده کند. الگوریتم مربوطه در قالب یک شی استراتژی ساخته شده و یک شی Executor را با آن پیکربندی می‌کند. می‌توان الگوریتم های مختلفی را زیر استراتژی اضافه کرد و تغییرات به قسمت‌های مختلف سیستم منتشر نمی‌شود. اجراکننده نیز الگوریتم مربوطه را از طریق پروکسی و بروکر اجرا کرده و تعامل بین زیرسیستم‌ها را برقرار می‌کند. این الگوریتم مشخص می‌کند که از چه سرویس‌هایی و به چه ترتیب باید استفاده کرد تا درخواست مشتری پاسخ داده شود. در نهایت پاسخ را به مشتری بازمی‌گرداند. در این ساختار می‌توانیم الگوهای Observer، Strategy، Broker، Microkernel و Remote Proxy را به وضوح ببینیم.



شکل ۱-۶: نمودار کلاس معماری پیشنهادی

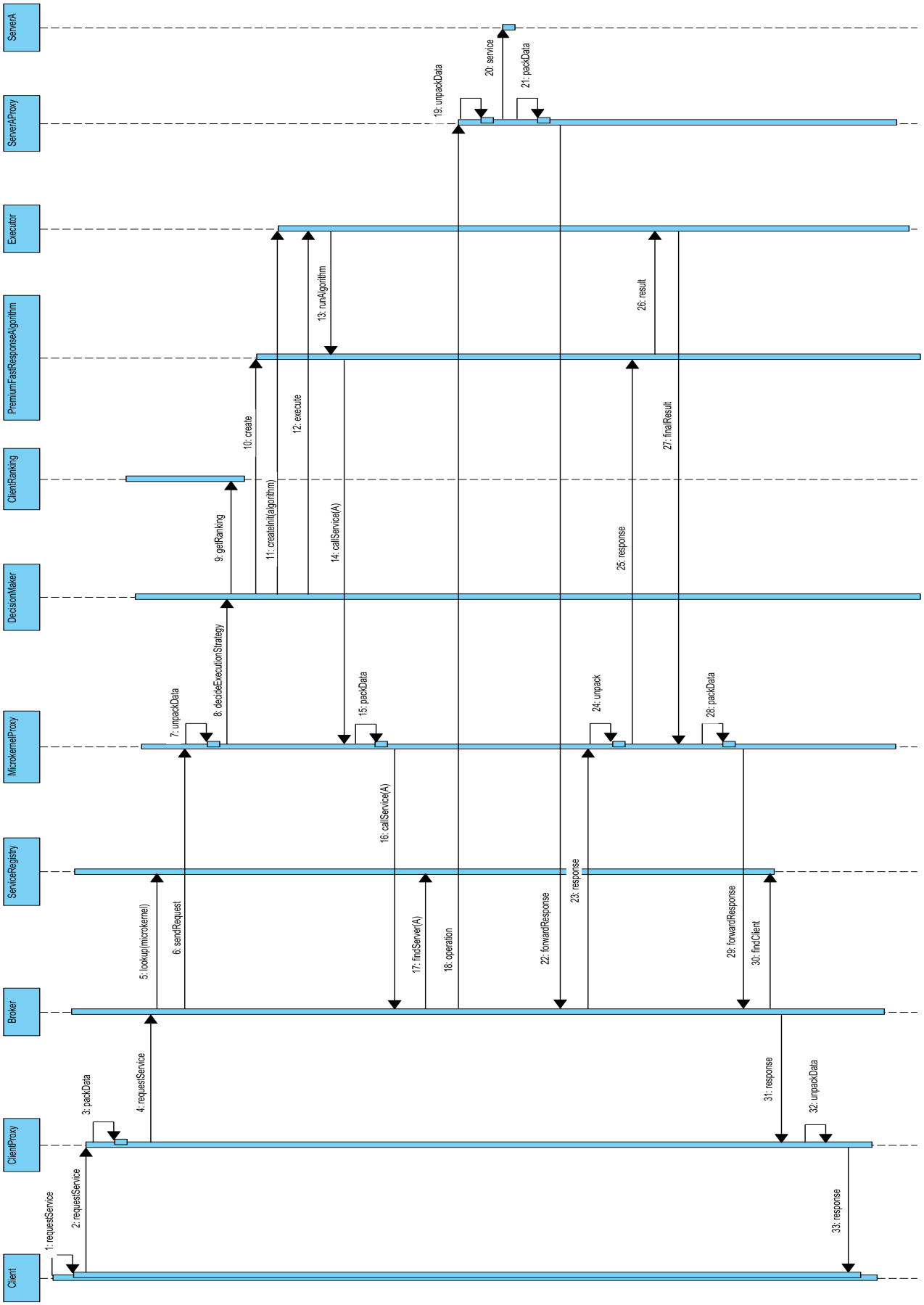
۱-۲-۶ رفتار راه‌حل

در این قسمت، رفتار راه‌حل پیشنهادی را در قالب چند سناریو مشاهده می‌کنیم.

۱-۲-۶-۱ سناریو اول

در شکل ۱-۷ نمودار توالی سناریو اول مشاهده می‌شود.

سناریو اول موقعیتی را نشان می‌دهد که کاربر دارای رتبه‌بندی Premium بوده و در اجرای درخواست آن کاربر، سیستم مجاز است تا از مسیرهای بهینه‌تر از نظر زمانی سعی در پاسخ درخواست کاربر کند. هرچند که این الگوریتم ممکن است منابع بیشتری را مصرف کند، ولی به دلیل رتبه‌بندی بالای کاربر مجاز است. کاربر ابتدا عملیات RequestService خود را فراخوانی می‌کند که در آن، درخواست به پروکسی نزد خودش واسپاری می‌شود. پروکسی درخواست کاربر را دریافت کرده، دیتا را بسته‌بندی و به بروکر ارسال می‌کند. در این حین جزئیات ارتباط شبکه‌ای با بروکر را پروکسی انجام می‌دهد. بروکر درخواست را دریافت کرده، از طریق رجیستری سرویس‌ها، مایکروکرنل را پیدا می‌کند و درخواست را به آن ارسال می‌کند. پروکسی سمت مایکروکرنل نیز درخواست را دریافت کرده، از بسته‌بندی خارج کرده و عملیات decideExecutionStrategy را از مولفه تصمیم‌گیر فراخوانی می‌کند. این عملیات نیز اطلاعات مربوط به رتبه‌بندی کاربر را با استفاده از شناسه کاربر از مولفه رتبه‌بندی دریافت می‌کند. اطلاعات مربوط به وضعیت فعلی منابع سیستم و سرویس درخواستی را نیز دارد و با تجمیع این اطلاعات، یک شی استراتژی از نوع الگوریتم PremiumFastResponse ساخته و یک executor نیز ساخته و آن را با استراتژی پیکربندی می‌کند و عملیات execute را فراخوانی می‌کند. اجرا کننده نیز شروع به اجرای الگوریتم و توالی فراخوانی سرویس‌ها می‌کند که البته در این قسمت صرفا نام سرویس‌ها بیان شده و ترجمه و یافتن آدرس فیزیکی توسط بروکر انجام می‌شود. پس درخواست‌ها را به ترتیب الگوریتم، به پروکسی ارسال کرده، پروکسی نیز به بروکر ارسال می‌کند. بروکر نیز سرویس مربوطه را از رجیستری یافته، درخواست را به آن محول می‌کند. سرویس نیز پس از اجرا و پاسخ به درخواست، آن را به بروکر باز میگرداند. بروکر نیز آن را به پروکسی مایکروکرنل و آن نیز به مولفه اجرا کننده بازمیگرداند. در نهایت مولفه اجرا کننده پاسخ نهایی را ساخته و به بروکر ارسال می‌کند. بروکر نیز پروکسی مشتری را یافته و پاسخ نهایی را به آن ارسال می‌کند. البته در نمودار توالی به دلیل صرفه‌جویی در فضای کاغذ، صرفا یکی از فراخوانی‌های سرویس‌ها نمایش داده شده است ولی در عمل چندین بار سرویس‌های متعدد فراخوانی می‌شوند که نمودار آن تکراری می‌شود.



شکل ۱-۷: نمودار توالی سناریو اول

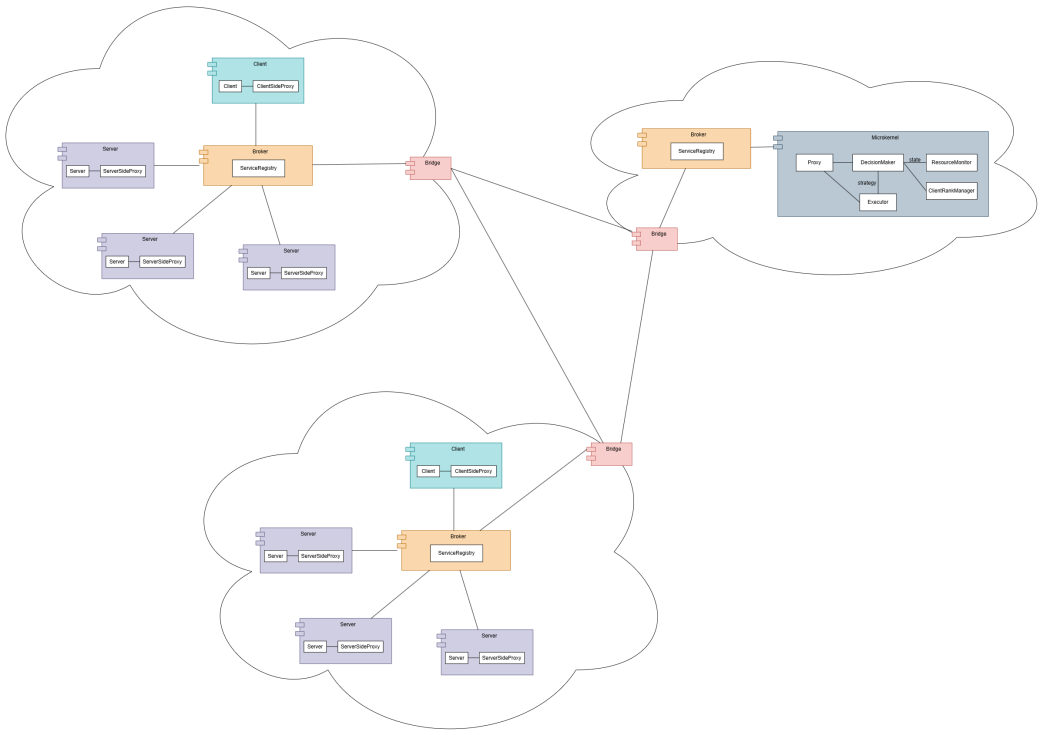
۱-۲-۶-۲ سناریو دوم

در شکل ۱-۹ نمودار توالی سناریو دوم مشاهده می‌شود.

در این سناریو، سیستم دچار کمبود و فشار در منابع ذخیره‌سازی می‌شود و مولفه پایش منابع با مشاهده وضعیت جدید، به تمام Observer های خود پیغام update می‌فرستد که مولفه DecisionMaker نیز به عنوان یک آبزرور، پیغام آپدیت را دریافت می‌کند و سپس به سراغ مولفه پایش منابع رفته و حالت فعلی را دریافت می‌کند. سپس یک مشتری درخواستی را ارسال می‌کند. درخواست مطابق شیوه‌ای که در سناریو اول دیدیم، از مشتری به مولفه تصمیم‌گیر در مایکروکرنل می‌رسد. این مولفه نیز به دلیل فشار روی منابع ذخیره‌سازی، یک الگوریتم مناسب شرایط فعلی که Space-Optimized است، به عنوان استراتژی ساخته و یک executor را با آن پیکربندی می‌کند. مولفه اجراکننده نیز مطابق شیوه‌ای که در سناریو اول دیدیم، به اجرای الگوریتم تعامل پرداخته و پاسخ به مشتری بازمی‌گردد. بدین شکل به‌صورت پویا در زمان اجرا، بسته به شرایط فعلی منابع سیستم، الگوریتم مناسب اجرا شد.

۱-۲-۷ سیستم‌های توزیع شده نامتجانس

در صورت وجود سیستم‌های نامتجانس توزیع شده، ساختار کلی سیستم مانند شکل ۱-۸ خواهد بود.



شکل ۱-۸: نمودار مفهومی معماری پیشنهادی در حالت سیستم‌های توزیع شده نامتجانس

این حالت تفاوت زیادی با حالت قبل که توضیح داده شد ندارد. فقط مولفه مایکروکرنل در فضای شبکه‌ای با بروکر خود قرار داشته و سایر زیرسیستم‌ها و مشتری‌ها نیز در فضاهای شبکه‌ای مختلف قرار گرفته‌اند که هرکدام بروکر خود را دارند. ارتباط بین این سیستم‌های توزیع شده در شبکه‌های نامتجانس، از طریق مولفه‌های Bridge انجام می‌گیرد. بروکرهای محلی با بریج ارتباط گرفته، آن نیز با بریج ریموت ارتباط برقرار می‌کند و بدین شکل سرویس گیری انجام می‌شود. سایر موارد مشابه حالت قبل است که توضیح داده شد.

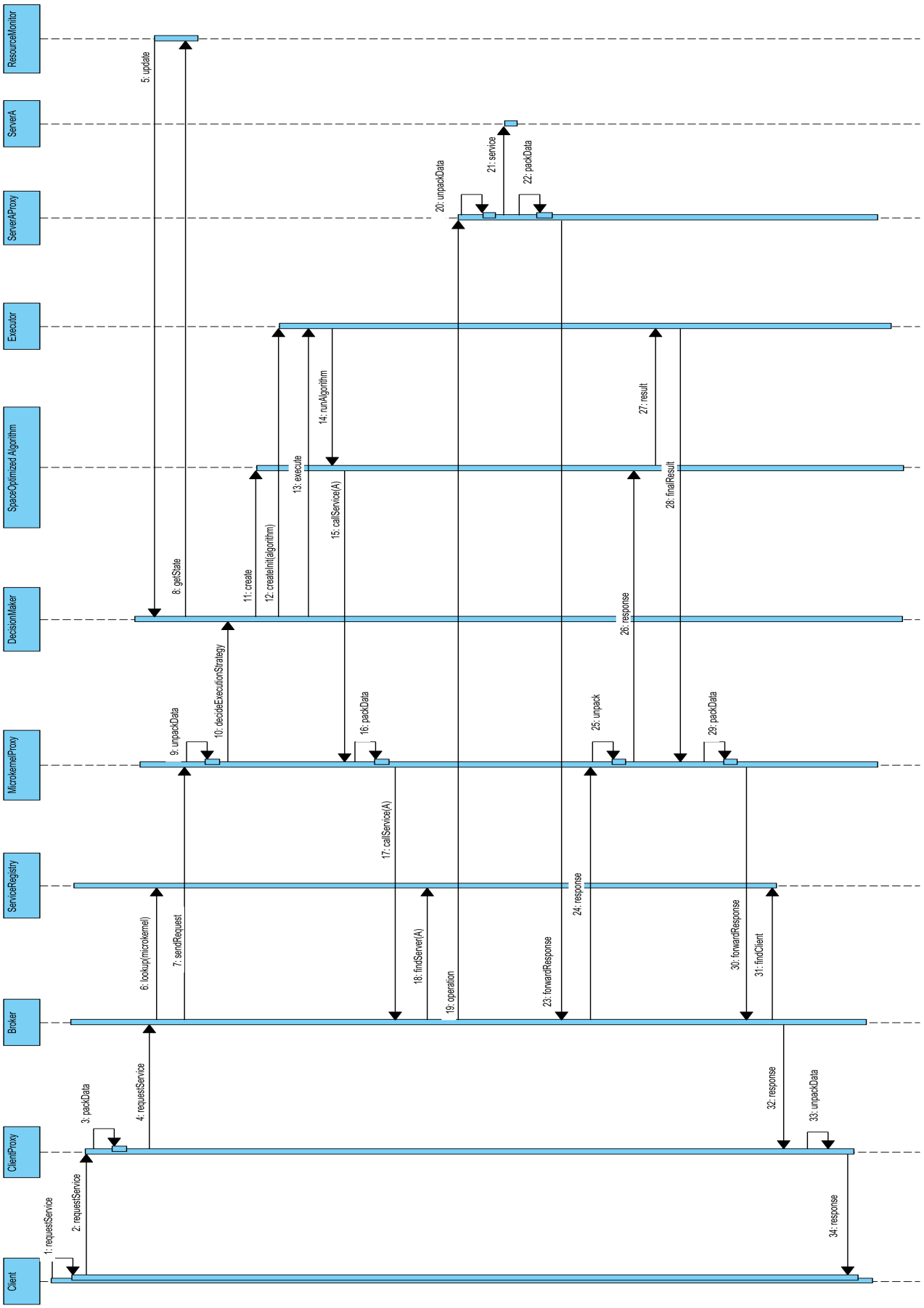
۱-۳ آزمون

در این قسمت توضیح می‌دهیم که هر کدام از زیرمسائل و نیازمندی‌های مطرح شده برای معماری سیستم هدف، چگونه توسط راه‌حل پیشنهادی محقق شده است.

توزیع‌شدگی مولفه‌های سیستم و تعامل توزیع شده مولفه‌ها: با استفاده از الگوی معماری Broker و الگوی طراحی پروکسی در راه‌حل پیشنهادی، مولفه‌های مختلف سیستم مانند مشتری‌ها، سرورها، مایکروکرنل و غیره می‌توانند به‌صورت توزیع شده و روی ماشین‌های مختلف اجرا شوند و location transparency وجود داشته و بروکر مسئول ترجمه، یافتن آدرس فیزیکی مولفه‌ها و ارتباط است. قسمت‌های مختلف سیستم اصلا نیازی به دانستن آدرس سایر قسمت‌ها و یا نحوه ارتباط با آن‌ها را ندارند. همچنین پروکسی‌های سمت مشتری و سرور نیز توهم محلی بودن سرویس را به آن‌ها می‌دهند و مشتری و سرور حتی دغدغه ارتباط با بروکر را نیز ندارند. در نتیجه ماجولاریتی خوبی برقرار بوده و توزیع شدگی نیز پاسخ داده شده است.

تطبیق و تغییر الگوریتم تعامل در زمان اجرا به‌صورت پویا: تمام وظیفه‌مندی و داده اصلی مربوط به الگوریتم تعامل در مایکروکرنل متمرکز شده است. مولفه DecisionMaker در مایکروکرنل، نقش اصلی انتخاب الگوریتم در زمان اجرا را بازی می‌کند. این مولفه دارای تمام اطلاعات و پارامترهای لازم برای تصمیم‌گیری شامل وضعیت فعلی منابع سیستم، رتبه‌بندی مشتری و سرویس درخواستی است و با این اطلاعات، در زمان اجرا به‌صورت پویا یک الگوریتم تعامل در قالب یک شی استراتژی ایجاد کرده و مولفه اجراکننده را با آن پیکربندی می‌کند. الگوریتم‌های مختلف نیز می‌توانند اضافه شوند و تغییرات به قسمت‌های مختلف سیستم منتشر نمی‌شود.

پایش و آگاهی از وضعیت کنونی منابع سیستم: با استفاده از الگوی Observer درواقع مولفه ResourceMonitor در نقش subject بوده، وضعیت منابع سیستم را پایش می‌کند و زمانی که تغییر عمده‌ای مانند کمبود فضای ذخیره‌سازی حاصل شود، پیغام update به تمام observer های خود ارسال می‌کند که مولفه DecisionMaker نیز یک آبزرور است. سپس مولفه تصمیم‌گیر getState کرده و حالت فعلی منابع سیستم را دریافت می‌کند و حالت درونی خود را مطابق آن بروز می‌کند. بدین شکل یکی از پارامترهای تصمیم‌گیری در رابطه با الگوریتم تعامل که وضعیت فعلی منابع سیستم بود، در اختیار مولفه تصمیم‌گیر قرار می‌گیر در زمان اجرا.



شکل ۱-۹: نمودار توالی سناریو دوم

آگاهی از رتبه‌بندی مشتریان و اعمال سیاست‌های مناسب: یک مولفه به نام ClientRankings مسئول نگهداری داده و عملیات مربوط به رتبه‌بندی کاربران است. مولفه تصمیم‌گیر، هنگام دریافت یک درخواست از سمت مشتری، با استفاده از شناسه مشتری، اطلاعات لازم مربوط به رتبه‌بندی مشتری را از این مولفه دریافت می‌کند و در فرایند تصمیم‌گیری الگوریتم تعامل دخیل می‌کند.

۱-۳-۱ ارزیابی

در این قسمت، توضیح مختصری از مزایا و معایت راه‌حل پیشنهادی بیان می‌شود.

۱-۳-۱-۱ مزایا

کاهش جفت‌شدگی[۷]: با به‌کارگیری الگوی Broker، جفت‌شدگی میان مولفه‌های توزیع شده بسیار کاهش می‌یابد. مشتری اصلا دغدغه یافتن سرویس و جزئیات سطح پایین ارتباطی ندارد و این بر دوش پروکسی است. location transparency وجود داشته و بروکر وظیفه یافتن آدرس فیزیکی و ارتباط با مولفه‌ها را دارد. مولفه‌ها نیز از طریق ارسال پیغام ارتباط برقرار می‌کنند. داخل مایکروکرنل نیز الگوریتم‌های تعامل در قالب استراتژی‌ها قرارگرفته و اجراکننده صرفا واسط را دیده و به زیرکلاس‌ها وابسته نیست. همچنین از الگوی Observer استفاده شده است برای انتشار تغییرات منابع سیستم با جفت‌شدگی پایین. در کل جفت‌شدگی در وضعیت مناسبی قرار دارد.

افزایش انسجام[۸]: دغدغه‌های ارتباط با مولفه‌ها در سیستم توزیع شده بر دوش بروکر و همچنین پروکسی‌ها است. مشتری دغدغه‌های مربوط به خود و هرکدام از سرورها نیز دغدغه‌های خود را دارند. وظیفه‌مندی و داده هسته سیستم نیز در مولفه مایکروکرنل قرارگرفته است. در داخل این مولفه، بخش‌های پایش منابع سیستم، تصمیم‌گیری استراتژی و الگوریتم تعامل و اجرای الگوریتم وجود داشته و به این شکل انسجام در قسمت‌های مختلف افزایش یافته است.

ساده‌تر شدن آزمون مولفه‌ها: به علت افزایش انسجام و کاهش جفت‌شدگی، آزمون‌پذیری در مولفه‌ها افزایش یافته است.

انعطاف‌پذیری: استفاده از الگوی Broker باعث شده است در سطح سیستم توزیع شده انعطاف‌پذیری داشته و امکان اضافه و کم کردن سرویس‌ها وجود داشته باشد. همچنین داخل مایکروکرنل، مولفه اجراکننده الگوریتم تعامل وابستگی به خود الگوریتم‌ها نداشته و می‌توان انواع الگوریتم‌ها را در زمان اجرا با آن پیکربندی

[۷] coupling
[۸] cohesion

و اجرا کرد. امکان گسترش الگوریتم‌ها نیز وجود داشته و تغییرات به‌صورت گسترده منتشر نمی‌شوند. وجود مولفه رتبه‌بندی مشتری نیز باعث افزایش انعطاف‌پذیری در تغییر رتبه‌بندی مشتری‌ها در زمان اجرا شده است.

جدایی دغدغه‌ها: دغدغه ترجمه آدرس فیزیکی و ارتباط در سطح سیستم توزیع شده بر دوش بروکر. دغدغه ارتباط با بروکر با پروکسی‌ها. مشتری و سرورها دغدغه‌های مربوط به خود را دارند. مایکروکرنل نیز دغدغه‌های وظیفه‌مندی هسته سیستم را داشته و داخل آن، مولفه‌های پایش منابع سیستم، تصمیم‌گیری الگوریتم، اجراکننده و رتبه‌بندی مشتری همگی مولفه‌های جدا بوده و دغدغه‌های مخصوص به خود دارند.

امکان replication: به‌دلیل استفاده از بروکر و توزیع شده بودن سیستم، برخی از بخش‌ها را می‌توان replicate کرد و در مواقع بروز خطا و اشکال در یک بخش، از replica آن می‌توان استفاده کرد.

و سایر مزایای الگوهای استفاده شده که مزایای خود را همراه خود آورده‌اند. الگوهایی مانند Broker، Observer، Strategy، Microkernel و Proxy.

۱-۳-۱-۲ معایب

کاهش کارایی: به دلیل افزایش غیرمستقیم بودن ارتباطات و ارسال و دریافت زیاد پیغام و همچنین افزایش تعداد مولفه‌های دخیل در سیستم و اضافه شدن انواع اشیای جعلی، همگی باعث کاهش کارایی از منظر حافظه و زمان می‌شوند.

افزایش تعداد اشیا: برای تحقق اهداف الگوها و معماری، انواع اشیای جعلی تعریف شده‌اند که در زمان اجرا با یکدیگر تعامل می‌کنند. تعداد زیادی پیغام نیز رد و بدل می‌شود. این افزایش در تعداد اشیا در سیستم باعث کاهش کارایی از منظر حافظه و زمان شده و روی منابع سیستم ممکن است فشار وارد کند.

عنصر مرکزی: در سیستم توزیع شده مولفه Broker مرکزی بوده و همچنین در سیستم قسمت مایکروکرنل که حاوی وظیفه‌مندی‌هایی نظیر پایش منابع سیستم، تصمیم‌گیری الگوریتم تعامل و غیره است، نقش عنصر مرکزی را بازی می‌کنند و معایب داشتن عنصر مرکزی را همراه خود دارند. مانند single point of failure بودن و خرابی عنصر مرکزی که سیستم را دچار مشکل می‌کند. یا تبدیل شدن به گلوگاه[۹] و کاهش کارایی و سایر مشکلات که بخاطر عنصر مرکزی بودن بوجود می‌آید.

افزایش احتمال خطا: به‌دلیل افزایش تعداد مولفه‌های دخیل در سیستم، تعداد بخش‌هایی که می‌توانند دچار خطا شوند نیز افزایش یافته است و این احتمال بروز خطا در سیستم را افزایش می‌دهد.

سنگین بودن مولفه تصمیم‌گیر: مولفه تصمیم‌گیر استراتژی و الگوریتم تعامل در داخل مایکروکرنل، باید پارامترهای مختلفی نظیر رتبه‌بندی و منابع فعلی سیستم را در نظر گرفته و برای الگوریتم تعامل تصمیم بگیرد

[۹] bottleneck

که این مولفه سنگین و پیچیده شده است.

دشواری پیدا کردن اشکال: به‌دلیل ماهیت توزیع شده بودن سیستم و افزایش تعداد مولفه‌های دخیل در سیستم، دنبال کردن و پیدا کردن منشا خطا در سیستم می‌تواند دشوار باشد.

پیچیدگی بالا: به‌دلیل افزایش تعداد مولفه‌ها و تعاملات بین آن‌ها در یک محیط توزیع شده، پیچیدگی کلی سیستم برای تحقق درخواست‌ها افزایش یافته است.

فصل ۲

معرفی دو Idiom

در این بخش، به معرفی دو Idiom در زبان جاوا می‌پردازیم. [۴]

۲-۱ Test Whether In Construction Phase

۲-۱-۱ نوع

این Idiom از نوع Object Creation and Initialization در زبان جاوا است.

۲-۱-۲ مسئله

گاهی می‌خواهیم بدانیم که در یک لحظه خاص، آیا برنامه مشغول ساخت یک نمونه جدید است یا نه.

- یک Class Invariant ممکن است در فاصله بین اجرای سازنده کلاس پدر و سازنده کلاس فرزند برقرار نباشد.
- بعضی متدها فقط باید از درون سازنده فراخوانی شوند، و برخی دیگر نباید هرگز از درون سازنده فراخوانی شوند.

۲-۱-۳ راه‌حل

برای هر کلاسی که بخواهیم بدانیم آیا برنامه مشغول ساخت یک نمونه جدید از آن است یا نه (و همچنین برای هر کلاس فرزند آن)، مراحل زیر را می‌توانیم انجام دهیم:

- یک فیلد به صورت زیر اضافه کنید:

```
private boolean myConstructionFinished = false;
```

- در انتهای سازنده (اما قبل از بررسی Class Invariant، اگر از آن استفاده می‌کنید) عبارت زیر را اضافه کنید:

```
myConstructionFinished = true;
```

- یک متد به صورت زیر اضافه کنید:

```
/** This method may also be protected or package-visible,
but it may not be private! */
public boolean constructionFinished()
{
        return myConstructionFinished;
}
```

اکنون می‌توانید با فراخوانی constructionFinished بررسی کنید که آیا نمونه فعلی (this) هنوز در حال ساخت است یا خیر. همچنین می‌توانید x.constructionFinished را صدا بزنید تا بفهمید آیا نمونه x در حال ساخت است یا نه.

دلیل اینکه این راه‌کار جواب می‌دهد:

- پیش از اجرای سازنده‌ها، تمام مقادیر بولین به‌صورت پیش‌فرض false هستند. (بر اساس Java Language Specification بخش‌های 15.8.1 و 4.5.4)
- سازنده‌ها به صورت زنجیره‌ای از بالا به پایین اجرا می‌شوند. بنابراین فیلد myConstructionFinished در کلاس‌های بالاتر زودتر به true تنظیم می‌شود.
- متد constructionFinished به صورت مجازی (virtual) است. بنابراین وقتی این متد صدا زده می‌شود، همیشه فیلد myConstructionFinished مربوط به پایین‌ترین کلاس (کلاسی که واقعا new شده) بررسی می‌شود. و این فیلد تنها در انتهای سازنده همان کلاس پایینی به true تنظیم می‌شود.

۲-۲ Avoid Final Strings For Unique Types

۲-۲-۱ نوع

این Idiom از نوع Type-Safety در زبان جاوا است.

۲-۲-۲ مسئله

در زبان جاوا، پیش از معرفی enum در نسخه 5، برنامه‌نویسان معمولا از public static final String برای تعریف مقادیر ثابتی که نقش "نوع محدود" را ایفا می‌کردند استفاده می‌کردند. این الگو برای موقعیت‌هایی که نیاز به تعداد محدودی مقدار مشخص (مثل جهت‌های جغرافیایی) داشتند به کار می‌رفت.

این Idiom به موضوع «EnumeratedTypesInJava» نزدیک است، اما به مشکلی ساده‌تر می‌پردازد و راه‌حل ساده‌تری را پیشنهاد می‌دهد.

در بسیاری از کدهای جاوا با الگوی رایج (و ضعیف) زیر مواجه می‌شوید:

```
۱  class Direction {
۲    public final static String NORTH = "North";
۳    public final static String SOUTH = "South";
۴    public final static String EAST = "East";
۵    public final static String WEST = "West";
۶
۷    public void setOrientation(String aHeading) {
۸      if (aHeading.equals(NORTH) { ... etc .. }
۹    }
۱۰
۱۱   public static void main(String args[]) {
۱۲         Direction dir = new Direction();
۱۳         dir.setOrientation(Direction.NORTH);
۱۴ }
```

هدف این کد، تعریف مقادیر خاص و منحصربه‌فرد برای پارامترهاست. شبیه به آنچه با define یا enum در زبان C انجام می‌دهند. اما مشکل این است که هر رشته‌ای را می‌توان به متد setOrientation پاس داد،

و تنها راه تشخیص صحت آن، بررسی در زمان اجرا است. همچنین استفاده از aHeading.equals هزینه‌بر است. البته می‌توان آن را با بررسی ارجاع مستقیم (مثل aHeading == NORTH) بهینه‌سازی کرد.

۳-۲-۲ راه‌حل

راه‌حل زیباتر در جاوای 1.1 و بعد از آن: استفاده از inner class

```
1  public class Direction {
2
3    // Define base type for constants
4    //
5    private static class Heading { };
6
7    public final static Heading NORTH = new Heading();
8    public final static Heading SOUTH = new Heading();
9    public final static Heading EAST = new Heading();
10   public final static Heading WEST = new Heading();
11
12   public void setOrientation(Heading aHeading) {
13         if (aHeading == NORTH || aHeading == SOUTH) {
14           System.out.println("Head for the pole!");
15         }
16   }
17   public static  void main (String argv[]) {
18         Direction dir = new Direction();
19         dir.setOrientation(Direction.NORTH);
20         dir.setOrientation(Direction.EAST);
21         dir.setOrientation(Direction.SOUTH);
22   }
23 }
```

اکنون بررسی نوع در زمان کامپایل انجام می‌شود و استفاده از نوع سفارشی خودتان، چندان پرهزینه‌تر از استفاده از رشته‌ها نیست. یک مشکل باقی می‌ماند: این متغیرهای final هیچ نمایش متنی ندارند.

اضافه کردن قابلیت نمایش متنی (toString):

```
private static class Heading {
  private String dir;
  Heading(String aDirection) {
    dir = aDirection;
  }
  public String toString() {
    return dir;
  }
}
public final static Heading NORTH = new Heading("NORTH");
public final static Heading SOUTH = new Heading("SOUTH");

public void setOrientation(Heading aHeading) {
  System.out.println("You are now Facing " + aHeading);
  if (aHeading == NORTH || aHeading == SOUTH) {
    System.out.println("Head for the pole!");
  }
}
```

Bibliography

[1] A. Shvets. https://refactoring.guru.

[2] J. R. V. J. Gamma Erich, Helm Richard. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[4] https://wiki.c2.com/?javaidioms.

Abstract

In this study, the first part examines the target system, extracts its requirements, proposes a solution by combining design and architectural patterns, and then evaluates the proposed solution. In the second part, two idioms in the Java programming language are introduced.

Keywords: design patterns, architectural patterns, architectural design, pattern combination, idiom

Sharif University of Technology
Department of Computer Engineering

Patterns in Software Engineering

Second Assignment

By:

Mehdi Eidi

Professor:

Dr. Raman Ramsin

Spring 2025