به نام خداوند بخشنده‌ی مهربان

دانشگاه صنعتی شریف

دانشکده‌ی مهندسی کامپیوتر

الگوها در مهندسی نرم‌افزار - تمرین دوم

استاد درس:

جناب آقای دکتر رامان رامسین

نگارش:

محمدمهدی فاریابی

۹۸۲۰۹۳۸۳

خرداد ماه ۱۴۰۱

فهرست مطالب

فهرست مطالب 2

مساله اول 4

بررسی مساله و اجزای آن 4

مساله 4

توزیع‌شدگی 4

تعاملی بودن 6

متمرکز بودن داده‌ها 6

نگهداری چندین نمایه از داده‌ها 7

پیچیدگی سرویس‌های پردازش و نمایش داده‌ها 7

توزیع‌شدگی سرویس‌ها روی سرورهای مختلف 7

نامرئی بودن موقعیت 8

محدود کردن پویا و آنی دسترسی 8

پایین بودن وابستگی 9

گسترش پذیری آسان سیستم و سرویس‌های پردازشی یا نمایشی 9

معماری پیشنهادی 10

چند نکته مهم 10

توصیف معماری 11

ساختار معماری 12

رفتار معماری 14

سناریوی ۱: محاسبه‌ی پیچیده 14

سناریوی ۲: به روزرسانی اطلاعات و رابط کاربری 17

معایب این معماری 20

مساله دوم 21

Use varargs judiciously 21

محیط 21

مساله 21

راهکار 23

مورد تامل 23

Return empty collections or arrays not nulls 24

محیط 24

مساله 24

راهکار 24

مورد تامل 25

منابع 26

مساله اول

بررسی مساله و اجزای آن

ابتدا به طرح کلی مساله می‌پردازیم، سپس با دقت به بررسی تک تک اجزای صورت سوال پرداخته و راهکارهای قابل تامل را بررسی و شرح می‌دهیم.

مساله

در یک سیستم تعاملی و توزیع شده[1]، داده‌ها در یک مخزن داده متمرکز[2] نگهداری می‌شود. چندین نمایه[3] مختلف از این داده‌ها به صورت همزمان به کاربرهای مختلف ارائه می‌شود. سرویس‌های پیچیده‌ای برای پردازش و نمایش داده‌ها به صورت توزیع‌شده روی چندین سرور و به صورت location-transparent (کاربرها متوجه توزیع‌شدگی و تفاوت موقعیت سرویس‌ها نمی‌شوند) به کاربرها ارائه می‌شود. مدیران سیستم باید بتوانند به صورت پویا دسترسی به بخش مشخصی از داده‌ها را محدود کنند. این محدودیت‌ها بایستی بلافاصله به نمایه‌های مرتبط با آن داده‌ها اعمال شود. این تغییرات ممکن است حتی باعث بسته‌شدن یک نمایه شود.
هدف، طراحی سیستمی است که در ضمن برآورده کردن نیازمندی‌های مطرح‌شده، وابستگی بین اجزای سیستم در آن به حدی کم باشد که بتوان سیستم و اجزای پردازش کننده/نمایش دهنده آن را به راحتی گسترش[4] داد.

اکنون به تفکیک به جزئیات، اجزا و جنبه‌های مساله پرداخته و در مورد هر یک و الگوهای احتمالی و نهایی فعال در آن به بحث و بررسی می‌پردازیم.

توزیع‌شدگی

در صورت پرسش بیان شده که این سیستم،سیستمی توزیع‌شده است. همچنین در ادامه نیز بیان شده که این سیستم، سیستمی تعاملی نیز هست. طبق آنچه از درس آموخته‌ایم بلافاصله چهار الگو برای مدیریت توزیع‌شدگی به ذهن می‌رسد.

1. الگوی معماری Microkernel که با استفاده از ایجاد یک مجموعه کوچک از functionality های پایه‌ای و ارائه باقی نیازمندی‌ها با استفاده از این مجموعه پایه‌ای سعی در حل مساله توزیع‌شدگی برای سیستم‌های با نیازمندی‌های دائما در

[1] interactive and distributed

[2] centralized

[3] view

[4] extension

حال تغییر دارد. از آنجایی که در این مساله صراحتا از چندین سرور برای سیستم استفاده شده و ذات مساله با توصیف ارائه شده از الگوی Microkernel مطابق نیست، از این الگو استفاده‌ای نخواهیم کرد.

2. الگوی معماری Pipes & Filters که با استفاده از ایجاد یک خط لوله[5] توزیع شده و موازی‌سازی پردازش[6] سعی در سریع حل کردن یک مساله‌ی توزیع‌شده همگن دارد. مساله‌ی ما (یک سیستم تعاملی با اجزای نه لزوما همگن) تفاوت ذاتی با فرضیات زمینه‌ای این الگو دارد و از این رو از این الگو نیز بهره‌ای نخواهیم برد.

3. الگوی معماری Broker که با استفاده از proxy های سمت کلاینت و سرور، service broker و همینطور bridge؛ مساله توزیع‌شدگی را به گونه‌ای حل می‌کند که اجزای مختلف از موقعیت یکدیگر خبر نداشته باشند و سرویس‌های با interface های مختلف بتوانند با یکدیگر به تعامل بپردازند. این الگو باعث می‌شود بتوانیم رفتارهای بیشتری را در قالب server side proxy و client side proxy به سیستم اضافه کنیم. همینطور ارتباطات بین سرویس‌های روی چند گره شبکه[7] را از طریق bridge به گونه‌ای به انجام برسانیم که هیچ وابستگی مستقیمی بین آنها برقرار نشود.

این الگو را در ترکیب با الگوی معماری PAC مورد استفاده قرار خواهیم داد. به این ترتیب که عامل[8]های الگوی معماری PAC برای ارتباط با سایر عامل‌های روی یک گره شبکه یا روی گره‌های دیگر شبکه از این الگو استفاده خواهند کرد. در ادامه مجددا در چند محل اشاره خواهیم کرد که به کار گیری این الگو، باعث برآورده شدن نیازمندی نامرئی بودن مکان[9] خواهد شد.

توجه کنیم از آنجایی که در PAC اینترفیس‌های عامل‌ها یکسان است. می‌توان به صورت خاص برای عامل‌های حاضر روی یک گره شبکه با اعمال ترکیبی الگوی طراحی Client-Dispatcher-Server با broker ارتباط مستقیم برقرار کرد و از سربار ناشی از واسط شدن broker برای برقراری ارتباط و احتمالا single point of failure شدن آن جلوگیری کرد. البته در ادامه‌ی حل سوال این فرض را در نظر نخواهیم گرفت.

4. الگوی معماری PAC که با استفاده از یک ساختار سلسله‌مراتبی از عامل‌های قائم به ذات، راهکاری برای رسیدگی به مساله تعامل با کاربر در یک سیستم تعاملی و رابط کاربری ارائه می‌کند. هر عامل باید داده و رفتار خود را نگهداری کند و کاملا coheisive باشد. در این ساختار سلسله‌مراتبی، عامل‌های سطح‌پایین (برگ‌ها[10]) وظیفه‌ی نمایش رابط تعاملی کاربری و اطلاعات را بر عهده داشته و متخصص این کار هستند. عامل‌های سطوح بالاتر وظیفه‌ی مدیریت دسته‌ای از عمل‌های سطح پایین را برای انجام چک‌های سطح بالا، هماهنگی داده‌ای و نمایش اطلاعات و رابط‌های

---

[5] pipeline

[6] parallel computing

[7] network node

[8] agent

[9] location transparency

[10] leaves

تعاملی مرکب، بر عهده دارند. عامل ریشه‌ای[11] نیز وظیفه‌ی نگهداری تمام داده‌ها و وضعیت سیستم را بر عهده دارد. باقی عامل‌ها برای دسترسی یا به روزرسانی داده‌ها، با عامل ریشه‌ای تعامل می‌کنند.

در این الگو،سیستم در حد اعلای انعطاف‌پذیری تصور می‌شود. تغییر منتشرشونده نداریم و سیستم می‌تواند به راحتی در طول زمان به تکامل رسیده و بزرگ شود.

در این الگو هر عامل دارای سه بخش اصلی است. بخش Control که یک واسط (تحقق الگوی طراحی Mediator) برای دو بخش دیگر و همینطور یک واسط برای ارتباط عامل‌های دیگر با این عامل است. بخش Presentation که وظیفه‌ی نمایش داده‌ها و ترسیم رابط کاربری را بر عهده دارد و متخصص این کار است. بخش Abstraction که وظیفه‌ی نگهداری داده‌های مرتبط با این عامل و همینطور حالت عامل را بر عهده دارد.

این الگو به دلیل ذات سلسله‌مراتبی و توزیع‌شده‌ی آن می‌تواند الگوی مناسبی برای طراحی سیستمی تعاملی و توزیع‌شده باشد. با توجه به توصیف مساله از توزیع‌شدگی سیستم تعاملی، استفاده از این الگو مناسب به نظر می‌رسد.

این الگو را در ترکیب با الگوی معماری Broker مورد استفاده قرار خواهیم داد. به این ترتیب که هر کدام از این عامل‌ها روی یک سرور (گره شبکه) قرار خواهند داشت و روی هر گره شبکه یک یا چند عامل حضور خواهند داشت. عوامل روی هر گره، ارتباطات با یکدیگر (شامل پیدا کردن یکدیگر و تعامل با یکدیگر) را از طریق broker در الگوی معماری Broker انجام خواهند داد.

تعاملی بودن

طبق آنچه از کلاس درس آموخته‌ایم برای ایجاد یک سیستم تعاملی توزیع‌شده می‌توانیم از الگوی معماری Presentation-Abstraction-Control یا PAC استفاده کنیم. توضیحات مربوط به این بخش را، به صورت جامع، در قسمت ۳ و ۴ از بخش قبل (توزیع‌شدگی) بیان کردیم.

متمرکز بودن داده‌ها

در صورت سوال بیان شده که داده‌های این سیستم تعاملی در یک مخزن، به صورت متمرکز و یک‌جا نگهداری می‌شوند. این توصیف منطبق بر نحوه‌ی نگهداری داده‌ها در الگوی معماری PAC برای ایجاد سیستم‌های تعاملی توزیع شده است. در این الگو تمامی داده‌های سیستم در عامل ریشه‌ای نگهداری می‌شود و سایر عامل‌ها برای دسترسی یا تغییر داده‌ها بایستی طی ساختار سلسه‌مراتبی، با بخش Control از عامل ریشه‌ای ارتباط برقرار کنند. از الگوی معماری PAC برای تحقق این جنبه از صورت سوال استفاده خواهیم کرد. به این ترتیب،عامل اصلی را تنها عامل دارای دسترسی به مخزن داده‌ها تعریف می‌کنیم. از الگوی

[11] root agent

طراحی remote proxy برای دسترسی به مخزن داده‌ها در عامل اصلی بهره می‌بریم. همینطور از آنجایی که احتمالا دسترسی به مخزن به لحاظ پردازشی و زمانی هزینه‌بر خواهد بود از در این proxy از caching هم استفاده خواهیم کرد (به نوعی الگوی طراحی lazy loading proxy).

نگهداری چندین نمایه از داده‌ها

طبق صورت سوال، فرض شده که چندین نمایه از داده‌ها نگهداری می‌شود. هر نمایه یک نمایش بصری از دادگان و احتمالا یک بخش از یک رابط کاربری یا یک رابط کاربری مجزا است. در هر صورت هر نمایه توسط یک سلسله‌مراتب از عامل‌ها در الگوی PAC نمایش داده می‌شود. برای نگهداری از چندین نمایه، استفاده از الگوی طراحی view handler مناسب به نظر می‌رسد. به این ترتیب که از یک view handler برای مدیریت چندین instance از کاربرد الگوی معماری PAC استفاده خواهیم کرد. با اعمال الگوی طراحی Observer روی مخزن داده‌ها، تغییرات رخ‌داده روی دادگان را به اطلاع view handler و همینطور تک‌تک عامل‌های ریشه‌ای در الگوی معماری PAC رسانده و از view handler برای مدیریت این view ها استفاده می‌کنیم.

پیچیدگی سرویس‌های پردازش و نمایش داده‌ها

در صورت سوال فرض شده سرویس‌های پردازشی و نمایشی داده‌ها، سرویس‌های پیچیده‌ای هستند. در این صورت احتمالا این‌سرویس‌ها برای سرویس‌دهی نیاز به تعامل با یکدیگر و اجرای یک الگوریتم پیچیده دارند. الگوی معماری PAC که در بخش قبل در مورد آن صحبت کردیم راهکاری برای مدیریت این پیچیدگی در سیستم‌های نمایشی داده‌ها (رابط کاربری) با ایجاد یک ساختار سلسه‌مراتبی ارائه می‌کند. اما برای مدیریت الگوریتم پیچیده در سرویس‌های پردازشی، فرض می‌کنیم که این الگوریتم نباید توزیع‌شده باشد (نگهداری و فهم راحت‌تر سیستم). از این رو استفاده از الگوی طراحی Facade برای کارگردانی بین سرویس‌ها در اینجا مناسب به نظر می‌رسد. ساختار کلی الگوریتم را در این Facade تعریف خواهیم کرد و کارگردان، برای انجام پردازش، تک‌تک اجزای کار را به سرویس‌های متناظر کارسپاری خواهد کرد. همچنین با فرض اینکه تعاملات این سرویس‌ها با هم نیز پیچیده باشد می‌توان از الگوی طراحی Mediator برای رفع پیچیدگی تعامل طرف سرویس‌ها استفاده کرد. همینطور دقت‌کنیم ما در این سیستم از الگوی معماری Broker و همینطور PAC استفاده می‌کنیم. استفاده از این broker در الگوی Broker و control در PAC خود مصداق استفاده از الگوی طراحی Mediator هستند.

توزیع‌شدگی سرویس‌ها روی سرورهای مختلف

همانطور که در بخش توزیع‌شدگی بیان کردیم، برای مدیریت توزیع‌شدگی در سرویس‌ها از الگوی Broker استفاده می‌کنیم. به این ترتیب سرویس‌ها فارغ از اینکه سرویس مخاطبشان روی گره شبکه مشترک با خودشان است یا اینکه در گره دیگری قرار دارد،

می‌توانند با آن سرویس از طریق Broker خود و سپس از طریق bridgeها ارتباط برقرار کنند. این مهم باعث عدم ایجاد وابستگی مستقیم[12] بین سرویس‌ها شده و همینطور موقعیت آنها را نیز از دید هم نامرئی می‌کند. استفاده از bridge در الگوی معماری Broker، مصداق استفاده از الگوی طراحی Bridge است. همین‌طور استفاده از broker نیز مصداق استفاده از الگوی طراحی Mediator برای برقراری ارتباط پیچیده است.

نامرئی بودن موقعیت

همانطور که در بخش قبل و همینطور بخش توزیع‌شدگی بیان کردیم، استفاده از الگوی معماری Bridge باعث ایجاد location transparency بین کلاینت و سرورها شده و پیچیدگی ارتباط در بستر شبکه را از دید آنها مخفی می‌کند. همینطور در بخش توزیع‌شدگی بیان کردیم که می‌توان برای سرویس‌ها/عامل‌های مستقر روی یک گره شبکه، از الگوی طراحی Client-Dispatcher-Server در ترکیب با broker استفاده کرد و مقداری از سربار ارتباطی کاست. این الگو هم با ارائه‌ی مکانیزمی مبتنی بر name address resolution، مانند broker باعث ایجاد موقعیت نامرئی می‌شود.

محدود کردن پویا و آنی دسترسی

در مورد محدود کردن پویای دسترسی با استفاده از اطلاعات ارائه شده در صورت سوال، دو فرض به ذهن می‌رسد.

1. دسترسی محدود شده برای یک بخش مشخص از داده‌ها، برای تمامی کاربرها اعمال می‌شود. در این صورت با استفاده از تعاریف الگوی معماری به‌کار گرفته‌شده‌ی PAC، تغییر در دسترسیِ مرتبط با داده‌ی مورد نظر در عامل ریشه‌ای باعث به روز شدن اطلاعات داخلی تمامی گره‌های مرتبط با آن اطلاعات شده و در نتیجه باعث به روز شدن (اعمال محدودیت در) نمایش اطلاعات خواهد شد. این پروسه شامل تعدادی فراخوانی در سلسله مراتب عامل‌ها در الگوی معماری PAC خواهد بود و نسبتا آنی صورت می‌پذیرد (از آنجایی که هر عامل یک رونوشت از اطلاعات مورد نیاز خود نگهداری می‌کند و ممکن است پیغام به‌روز رسانی مدت اندکی بعد از اعمال به روزرسانی به این عامل برسد). اگر بخواهیم این تغییرات بی هیچ تاخیری اعمال شوند باید هیچ کدام از عوامل، رونوشتی از اطلاعات مورد نیاز خود نگهداری نکنند و به ازای هر نیاز به دسترسی به اطلاعات، داده‌ی مورد نیاز خود را از طریق ساختار سلسله‌مراتبی، مستقیما از عامل ریشه‌ای دریافت کنند (در این صورت به روزرسانی دسترسی، به صورت آنی و بی هیچ تاخیری صورت می‌پذیرد؛ اما در عوض نمایش رابط کاربری تعاملی تاخیر[13] بیشتری خواهد داشت). از آنجایی که در صورت مساله صحبتی از اهمیت تاخیر پایین به عمل نیامده اما بر آنی بودن اعمال دسترسی‌ها تاکید شده،از مدل دوم استفاده خواهیم کرد.

---

[12] direct coupling

[13] latency

2. دسترسی محدود شده، برای کاربرانی که به دلخواه مشخص می‌شوند اعمال می‌شود. در این صورت باید در مسیر پردازش هر درخواست، اطلاعات کاربرِ درخواست‌دهنده هم منتقل شود. از آنجایی که فرض کرده‌ایم داده‌ها به صورت متمرکز در عامل ریشه‌ای نگهداری می‌شوند، این حالت دسترسی محدود شده هم تفاوتی با حالت قبل ندارد و مدیریت دسترسی در سطح عامل ریشه‌ای انجام خواهد شد. به صورت دقیق‌تر از از الگوی طراحی Access Control در سطح Proxy داده‌ها در عامل ریشه‌ای استفاده می‌شود.

از آنجایی که ما در این سیستم از الگوی طراحی View handler هم استفاده می‌کنیم، همینطور از آنجایی که View handler خود با استفاده از الگوی Observer بر تغییرات داده‌ها و دسترسی‌ها نظارت دارد، می‌تواند در اثر برخی تغییرات به کلی یک یا تعدادی از view ها (پنجره‌های رابط کاربری[14]) را نیز ببندد. از طرفی دیگر در مواردی ممکن است دسته‌بندی‌هایی از پیش‌تعریف‌شده برای دسترسی‌ها وجود داشته باشد که می‌توان با به کار بردن الگوی طراحی State و مرتبط کردن منطق هر یک از این دسته‌ها با یک state آن‌ها را اعمال کرد. در اینجا، با توجه به اینکه در صورت سوال اشاره‌ای به این موضوع نشده و برای سادگی بیشتر از این حالت صرف نظر می‌کنیم.

پایین بودن وابستگی

استفاده از الگوهای طراحی Broker (بخش broker و bridge)، الگوی PAC (بخش control)، الگوی Proxy، الگوهای Facade، Observer و Mediator، همگی و همگی باعث ایجاد indirect coupling شده و وابستگی مستقیم بین عناصر محیط را از بین می‌برد. صرفا عناصری که وابستگی ذاتی به یکدیگر دارند مثل Proxy و Object متناظرش با یکدیگر coupled می‌مانند که ایرادی ندارد و باعث ایجاد تغییر منتشر شونده نخواهد شد. این پایین بودن وابستگی و بالا بودن cohesion ناشی از اعمال الگوهایی مانند PAC، سبب می‌شود که این سیستم، مصداق برآورده شدن اصل OCP[15] در شی گرایی باشد.

گسترش پذیری آسان سیستم و سرویس‌های پردازشی یا نمایشی

همانطور که در بخش قبل بیان شد. این سیستم می‌تواند مصداقی از برآورده شدن اصل OCP در شی گرایی باشد. در واقع یعنی گسترش سیستم باعث ایجاد نیاز به تغییرات نامرتبط و به خصوص تغییرات منتشر شونده نخواهد شد. این تعریف گسترش‌پذیری آسان است.

---

[14] user interface windows

[15] Open Closed Principle (Open to extension and Closed to modification)

معماری پیشنهادی

در این بخش ابتدا فرضیاتی که در حل مساله در نظر گرفته‌ایم را شرح داده و سپس به شرح معماری پیشنهادی و اجزای آن می‌پردازیم. تلاش می‌کنیم با استفاده از نمودارهای کلاس[16] و توالی رخداد[17]، نسبت به ساختار و رفتار[18] معماری پیشنهادی در سناریوهای مختلف شهود کافی را منتقل کنیم. در نهایت معماری‌ای که ارائه کرده‌ایم را مورد نقد قرار داده و تلاش می‌کنیم از معایب آن سخن بگوییم.

چند نکته مهم

1. طبق صورت سوال موجودیت‌های ادمین (Admin) و کلاینت خارجی (External Client) مورد تصور هستند و می‌توان نمودار کلاسی آن‌ها را ترسیم و ارتباطات آن‌ها با سایر اجزا را ترسیم کرد. از آنجایی که این موجودیت‌ها هم در یکی از گره‌های فیزیکی حضور داشته و ارتباط آن‌ها با اجزا مشابه سایر اجزای سیستم است، برای سادگی از ترسیم آن‌ها صرف‌نظر کرده‌ایم.
2. در یکی از نمودارهای توالی، ارتباطات بین اجزای الگوی Broker را ترسیم‌کرده‌ایم و در ادامه از ترسیم این ارتباطات برای سادگی بیشتر ترسیم و فهم نمودارها صرف‌نظر کرده‌ایم. بدیهی است که تمام اجزای این سیستم، توزیع‌شده هستند و ارتباطات آنها نیز طبق قواعد الگوی معماری Broker برقرار خواهد بود.
3. برای موجودیت‌های DataSupplier ، ConcreteView و DataSupplierProxy، تناظر تعریف‌شده‌ای در تعامل با الگوی Broker در نظر نگرفته‌ایم. بدیهی است که این موجودیت‌ها هم مطابق سایر موجودیت‌های سیستم در تعامل از الگوی Broker و از قواعد مشابه ساختاری و رفتاری پیروی می‌کنند.
4. در نمودار کلاسی برای اختصار و صرفه‌جویی در فضا گاهی نام attribute ها و method ها در یک خط و در کنار هم و گاهی به صورت ترکیبی آمده (به عنوان مثال[set/getData]).. چنین ترکیبهایی بیانگر هر دو نام برای method یا attribute کلاس مورد نظر هستند.
5. در method های بخش Presentation از Agent های الگوی PAC، به جای method های move، zoom، update و print، برای اختصار از method جایگزین draw استفاده شده است.

---

[16] class diagram

[17] sequence diagram

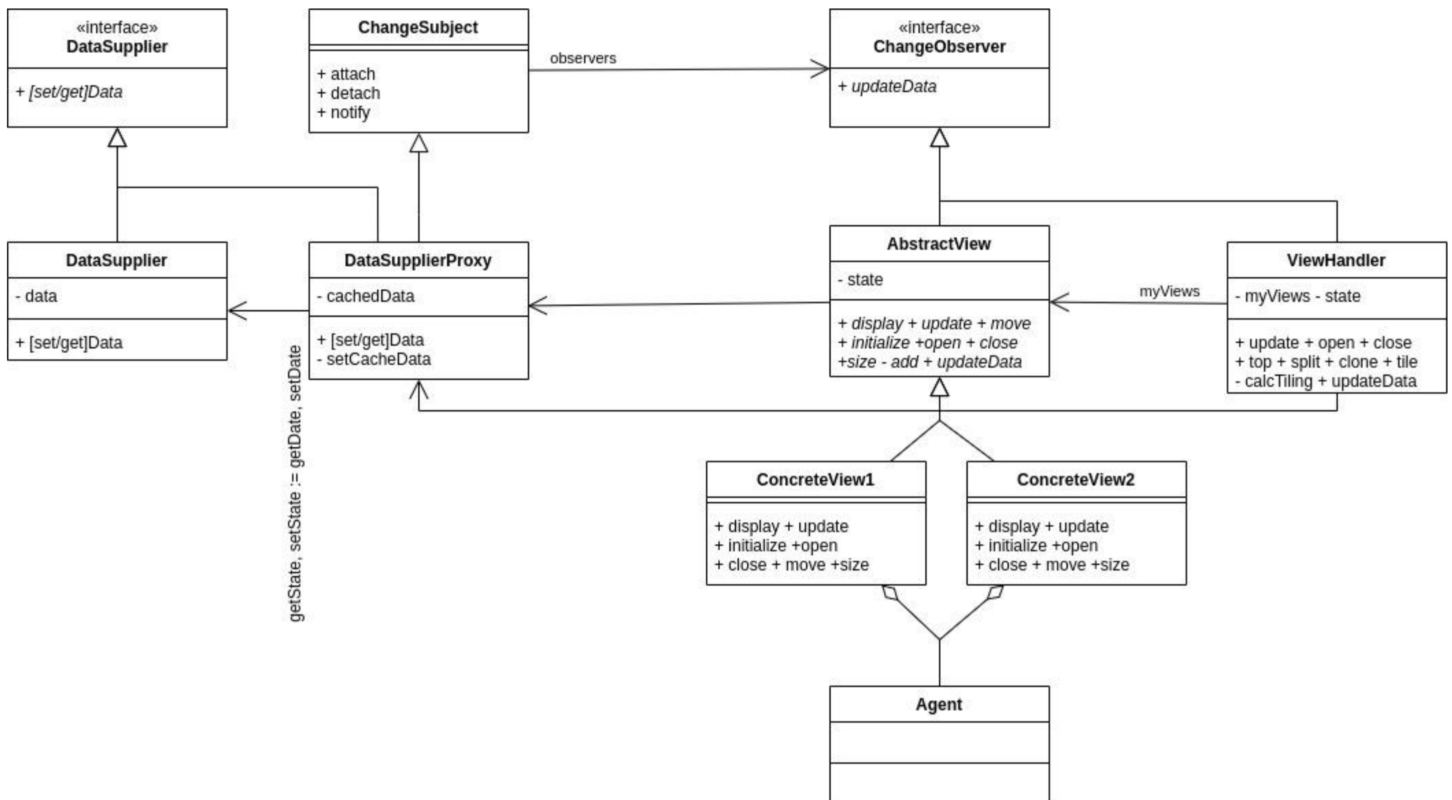
[18] structure and behavior

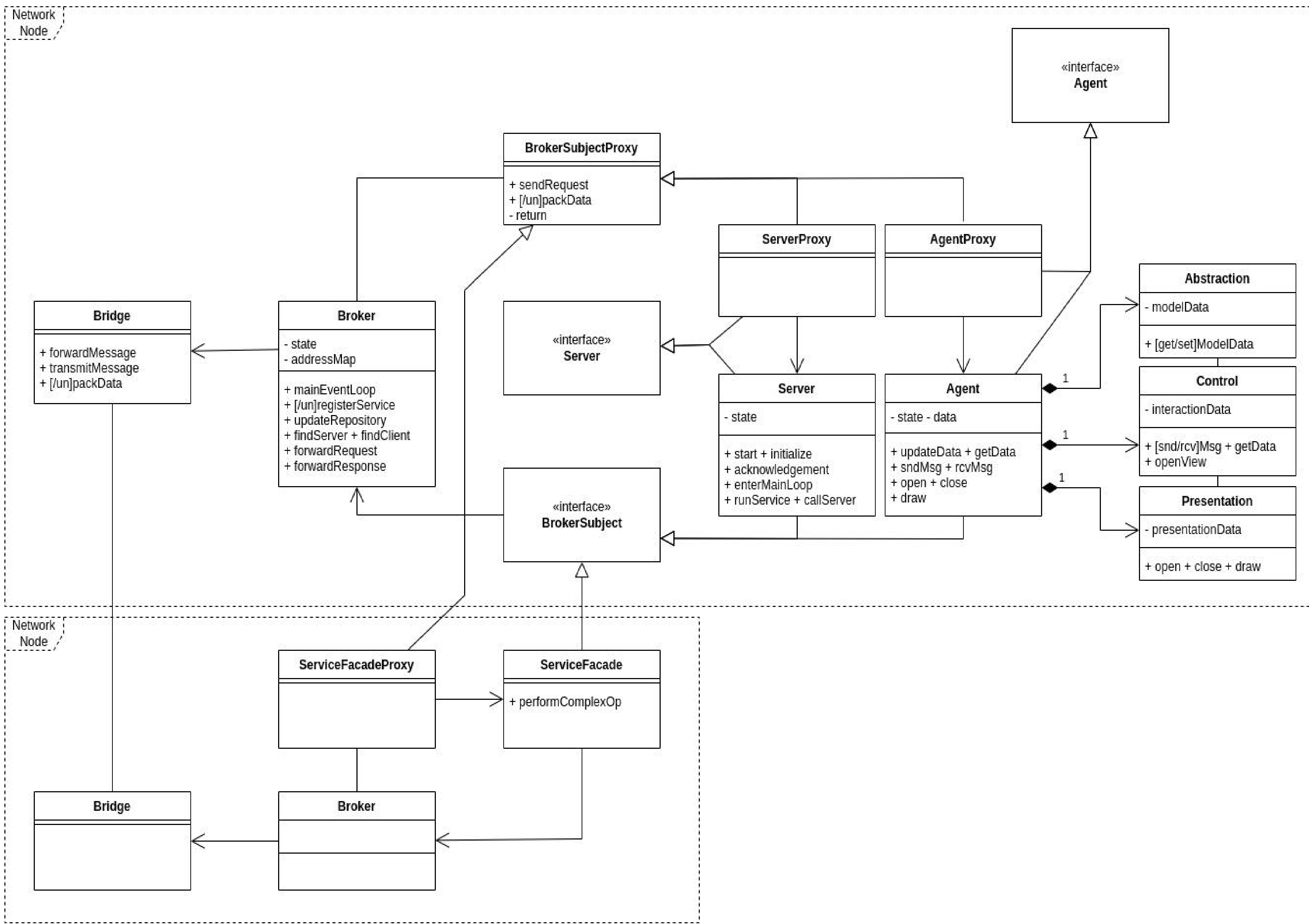
توصیف معماری

برای این سیستم از ترکیب الگوهای PAC و Broker بهره می‌بریم. تمامی اجزای سیستم طبق سوال می‌توانند توزیع‌شده و دور از یکدیگر باشند. فرض می‌کنیم این اجزا شامل عامل‌های الگوی PAC، سرویس‌های پردازشی سیستم، کارگردان‌ها، مخزن داده، کلاینت‌ها، ادمین‌ها و ویوها باشند. برای ارتباط بین اجزا از الگوی Broker استفاده می‌کنیم. به این ترتیب به نحوی که موقعیت عناصر برای یکدیگر نامرئی باشد، ارتباط بین آنها میسر می‌شود. هر ویو یک رابط کاربری یا بخش از رابط کاربری است که با استفاده از الگوی PAC ساخته شده و می‌تواند در اختیار یک یا چند کاربر قرار گیرد. برای مدیریت ویوها از الگوی View Handler بهره می‌بریم. به این ترتیب این View Handler و تمامی View ها، مخاطب مخزن اطلاعاتِ متمرکز و یکجای ما خواهند بود. برای دسترسی به این مخزن اطلاعات از یک access management proxy با قابلیت caching استفاده خواهیم کرد. ادمین سیستم، دسترسی‌ها را در خود مخزن تعریف کرده و این proxy، داده را بعد از بررسی دسترسی در اختیار کاربر قرار می‌دهد. این proxy برای کاهش دسترسی به منبع گران‌قیمت storage، از یک مکانیزم caching برای کاهش خواندن‌ها و حتی نوشتارها بهره می‌برد (caching برای نوشتارها را برای سادگی بیشتر در نمودارها در نظر نگرفته‌ایم). هر یک از اجزای View با توجه به جواب دریافتی از درخواستِ دریافت داده‌شان، خواهند دانست که بخش مربوطه بایستی به‌روز شود یا اینکه حذف گردد. برای اطلاع رسانی به اجزای View ها از به‌روز شدن داده‌ها نیز از الگوی Observer استفاده کرده‌ایم. در نهایت، از آنجایی که فرایندهای پردازشی در سیستم پیچیده هستند، این فرایندها را از طریق یک Facade روی سیستم اجرا خواهیم کرد. دقت کنیم در این مساله هر یک از اجزا می‌توانند سرور و در عین حال کلاینت برای اجزای دیگر باشند. از این رو Proxy های الگوی Broker را Proxyهایی با قابلیت‌های کلاینت و سرور در نظر گرفته‌ایم.

ساختار معماری

با بررسی نمودار کلاس به ساختار این معماری می‌پردازیم. دقت کنیم در اینجا برای سادگی بیشتر نمودار، برای Proxy ها و مخاطبین در الگوی Broker، اینترفیس یکسان تعریف کرده‌ایم. همچنین در مواردی که یک کلاس برای راحتی ترسیم تکرار شده است، از تکرار attributeها و method های آن چشم‌پوشیده‌ایم.




Network Node
«interface»
Agent
BrokerSubjectProxy
+ sendRequest
+ [/un]packData
- return
ServerProxy
AgentProxy
Abstraction
- modelData
+ [get/set]ModelData
Bridge
+ forwardMessage
+ transmitMessage
+ [/un]packData
Broker
- state
- addressMap
+ mainEventLoop
+ [/un]registerService
+ updateRepository
+ findServer + findClient
+ forwardRequest
+ forwardResponse
«interface»
Server
Server
- state
+ start + initialize
+ acknowledgement
+ enterMainLoop
+ runService + callServer
Agent
- state - data
+ updateData + getData
+ sndMsg + rcvMsg
+ open + close
+ draw
1
Control
- interactionData
+ [snd/rcv]Msg + getData
+ openView
1
1
Presentation
- presentationData
+ open + close + draw
«interface»
BrokerSubject
Network Node
ServiceFacadeProxy
ServiceFacade
+ performComplexOp
Bridge
Broker

رفتار معماری

برای درک بیشتر این معماری به بررسی دو سناریوی جامع می‌پردازیم. یکی تعاملات انجام شده برای محاسبه‌ی یک عملیات پیچیده از طریق کارگردان و دیگری به روز شدن اطلاعات سیستم و به‌روز شدن نمایش رابط کاربری. به تشریح هر دو می‌پردازیم.

سناریوی ۱: محاسبه‌ی پیچیده

در این سناریو کلاینت درخواست اجرای یک محاسبه‌ی پیچیده را روی اجزای سیستم صادر کرده است. این درخواست به یک شی از نوع سرویس کارگردان[19] می‌رسد. فرایند اینگونه دنبال می‌شود:

1. facade به عنوان کلاینت سرویسی که می‌خواهد فراخوانی کند، متد callService خود را اجرا می‌کند. اجرای این متد باعث فراخوانی sendRequest روی پروکسی[20] مربوط به facade می‌شود (پراکسی در نقش کلاینت).
2. پروکسی facade متد packData را فراخوانی کرده و داده‌های درخواست را آماده‌ی ارسال به broker می‌کند. سپس با فراخوانی متد sendRequest از broker، درخواست کلاینت (کارگردان) را به سمت سرور مخاطب ارسال می‌کند.
3. broker حاضر در گره شبکه‌ی کارگردان، با اطلاعات دریافت شده و با فراخوانی متد findServer از خود، به دنبال محل سرور مورد نظر می‌گردد، سپس اطلاعات درخواست و آدرسی که پیدا کرده را به bridge مستقر در گره خود می‌فرستد.
4. bridge حاضر در گره مبدا اطلاعات درخواست و آدرس را با فراخوانی متد packData آماده‌ی ارسال به bridge حاضر در گره مقصد می‌کند و سپس با فراخوانی متد transmitMessage آن bridge، اطلاعات را ارسال می‌کند.
5. bridge حاضر در گره مقصد با فراخوانی متد unpackData درخواست دریافتی را باز کرده و با فراخوانی رویه forwardMessage اطلاعات را برای broker هم‌گره خود می‌فرستند.
6. broker مقصد با استفاده از اطلاعات دریافتی از bridge خود و با فراخوانی متد findServer موقعیت سرور مقصد هم‌گره خود را پیدا کرده و با فراخوانی متد callService از پروکسی آن سرویس، درخواست را به آن پراکسی می‌رساند.
7. پروکسی سرویس مقصد با استفاده از متد unpackData، اطلاعات درخواست دریافتی را آماده کرده و با فراخوانی متد runService، درخواست را برای سرویس مقصد می‌فرستد.
8. سرویس مقصد درخواست را اجرا می‌کند. دقت کنیم این اجرای درخواست می‌تواند خود با فراخوانی‌های شبکه‌ی متعدد و تودرتو[21] همراه باشد. سپس پاسخ درخواست را به پروکسی خود باز می‌گرداند.
9. پراکسی سرویس مقصد، پاسخ دریافتی را با استفاده از متد packData آماده کرده و با فراخوانی متد forwardResponse از broker هم گره خود، برای آن می‌فرستد.

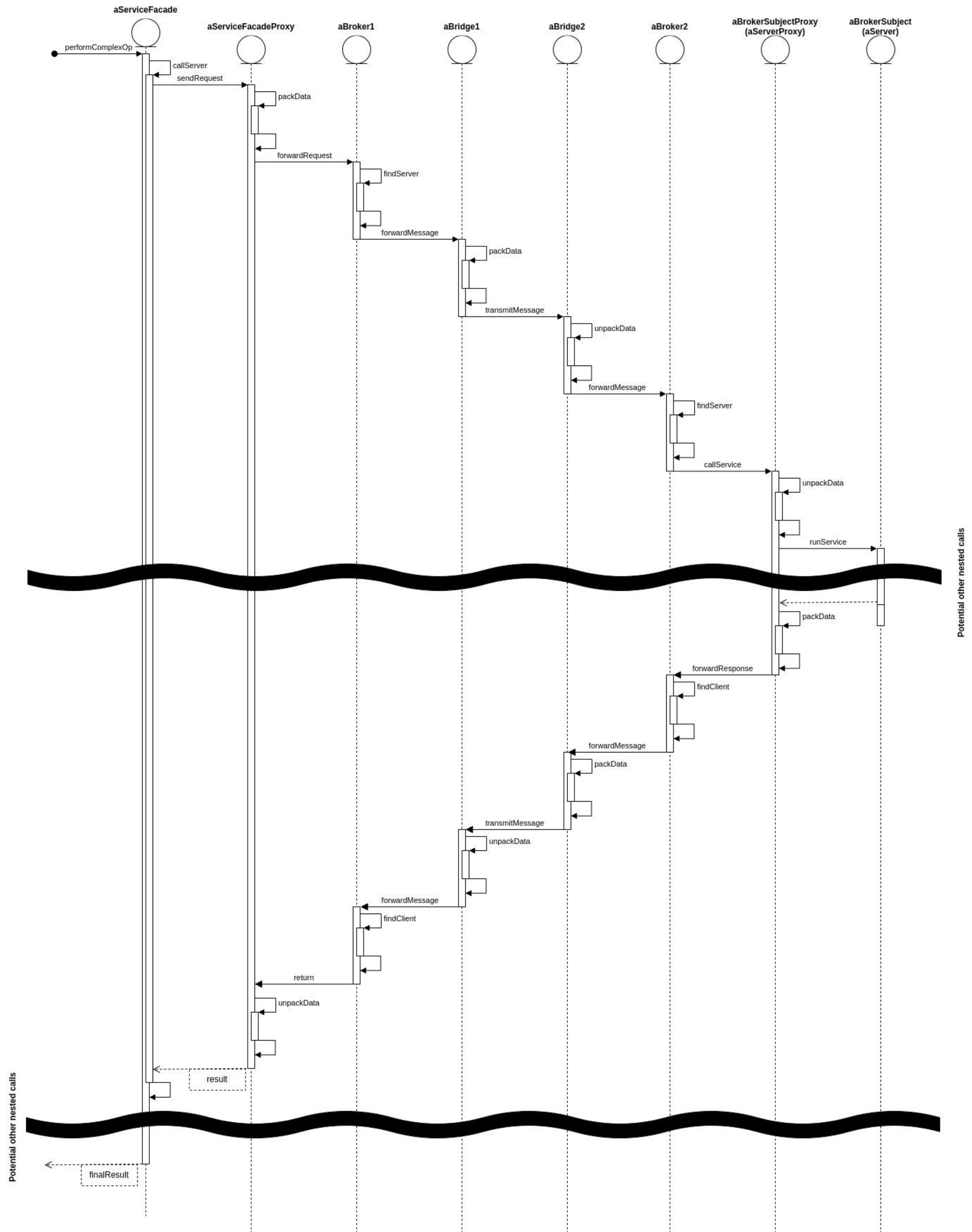
[19] facade

[20] proxy

[21] nested

10. broker مقصد اطلاعات دریافتی از پروکسی سرور مقصد را دریافت و با استفاده از متد findClient، موقعیت سرور درخواست‌دهنده در شبکه را پیدا می‌کند (سرور کارگردان). سپس اطلاعات پاسخ و آدرس محاسبه شده را با استفاده از فراخوانی رویه forwardMessage به bridge خود می‌فرستد.
11. bridge گره مقصد با متد packData، اطلاعات را آماده ارسال کرده و با فراخوانی متد transmitMessage به bridge گره مبدا تحویل می‌دهد.
12. bridge گره مبدا با استفاده از متد unpackData، داده‌ی دریافتی را باز کرده و با فراخوانی متد forwardMessage به broker خود تحویل می‌دهد.
13. broker مبدا با استفاده از متد findClient، درخواست‌دهنده‌ی پاسخ را پیدا کرده و با استفاده از فراخوانی متد return از پراکسی درخواست‌دهنده، اطلاعات دریافتی را برای آن پراکسی ارسال می‌کند.
14. پروکسی درخواست دهنده با استفاده از متد unpackData، اطلاعات دریافتی را باز کرده و در نهایت پاسخ را به درخواست دهنده (در اینجا سرویس کارگردان) می‌رساند.
15. سرویس کارگردان در اینجا مجددا با طی مراحل ۱ تا ۱۴ به دفعات، ادامه‌ی محاسبات را با کارچرخانی کردن بین عناصر مختلف سیستم ادامه می‌دهد.
16. در نهایت پاسخ نهایی به درخواست کننده‌ی کارگردانی برگردانده می‌شود.

توجه کنیم که درخواست کارگردانی هم از طریق تعاملات در شبکه و با الگوی Broker به دست کارگردان رسیده است.


aServiceFacade
aServiceFacadeProxy
aBroker1
aBridge1
aBridge2
aBroker2
aBrokerSubjectProxy
(aServerProxy)
aBrokerSubject
(aServer)
performComplexOp
callServer
sendRequest
packData
forwardRequest
findServer
forwardMessage
packData
transmitMessage
unpackData
forwardMessage
findServer
callService
unpackData
runService
Potential other nested calls
packData
forwardResponse
findClient
forwardMessage
packData
transmitMessage
unpackData
forwardMessage
findClient
return
unpackData
result
Potential other nested calls
finalResult

سناریوی ۲: به روزرسانی اطلاعات و رابط کاربری

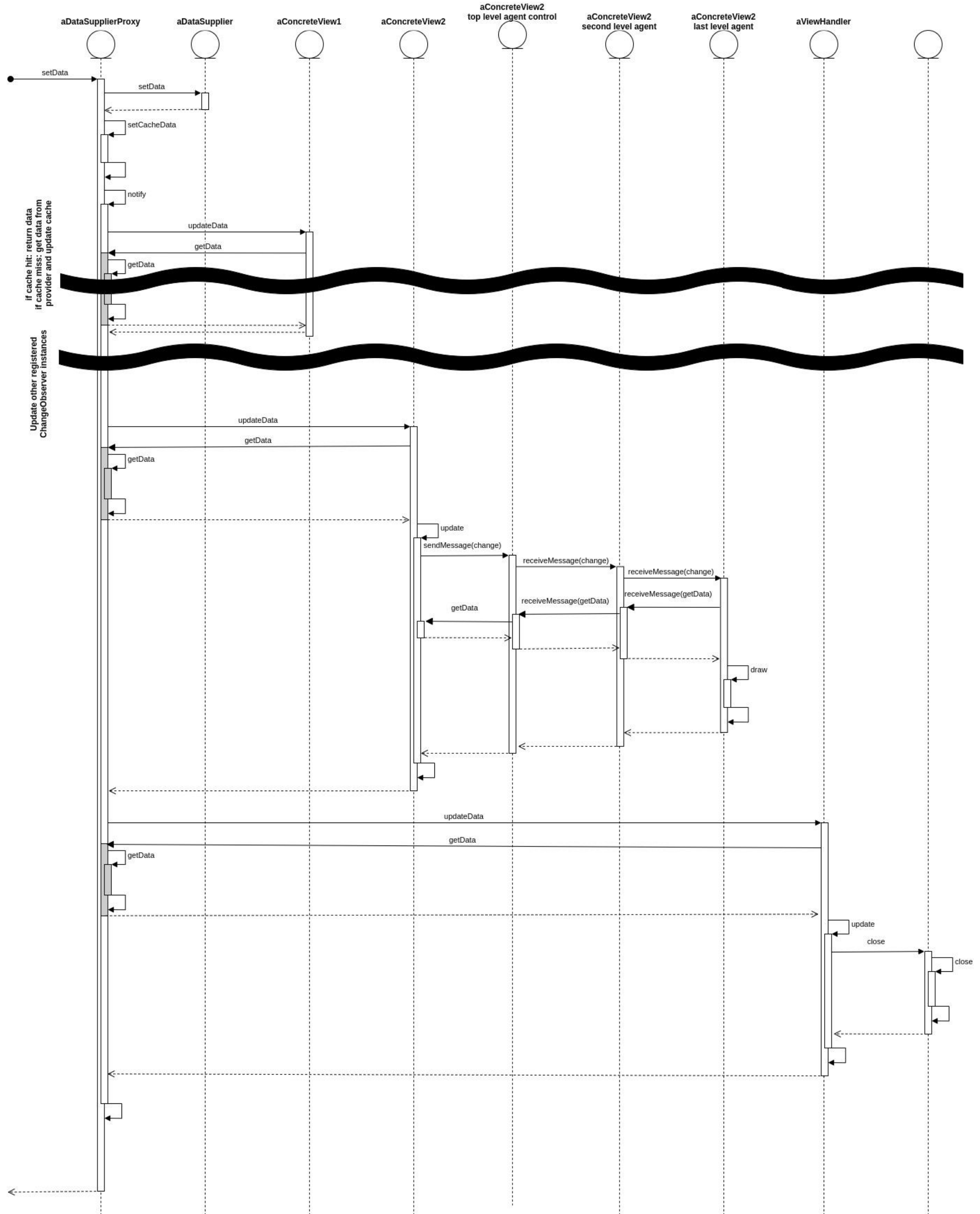
در این سناریو، یکی از بخش‌های سیستم، کلاینت و یا ادمین سیستم، اطلاعات یا دسترسی‌های سیستم را مورد تغییر قرار می‌دهد. فرایند به روزرسانی نمایش اطلاعات در سیستم اینگونه دنبال می‌شود:

1. متد setData از پراکسی داده فراخوانی می‌شود.
2. پراکسی داده متد setData از مخزن داده را فراخوانی می‌کند. سپس متد setCacheData را برای به‌روز رسانی اطلاعات حافظه‌ی کش[22] فراخوانی می‌کند.
3. پراکسی داده با فراخوانی متد notify خود، مطابق الگوی Observer، فرایند اطلاع‌رسانی به تمام موجودیت‌های observer خود را شروع می‌کند.
4. این فراخوانی باعث می‌شود متد updateData هر یک از این observerها فراخوانی شود. هر observer در صورت نیاز، متد getData از پراکسی داده را فراخوانی می‌کند.
5. فراخوانی متد getData از پراکسی داده باعث می‌شود در صورتی که داده‌ی مورد انتظار در کش این پراکسی وجود داشت، درخواستی به مخزن داده فرستاده نشود و داده‌ی موجود، خروجی داده شود. در صورتی که داده درکش وجود نداشت، درخواست منجر به دریافت داده از مخزن، قرار گرفتن داده در کش و سپس برگرداندن داده به درخواست‌کننده می‌شود.
6. تمام observerها به ترتیب با فراخوانی updateData از تغییرات مطلع می‌شوند. حالتی را تصور می‌کنیم که observer ما یک View در الگوی PAC است. در واقع با یک عامل ریشه‌ای در الگوی PAC طرف هستیم. این عامل با فراخوانی getData از پراکسی داده، داده‌ی مورد نیاز را دریافت می‌کند.
7. سپس با فراخوانی متد update خود، به فرزندان مستقیم خود در توالی، با استفاده از متد sendMessage پیغام تغییر را ارسال می‌کند. این پیغام تغییر با استفاده از متد receiveMessage در سلسه‌مراتب پایین می‌رود تا به عامل‌های پایین‌دستی برسد.
8. هر یک از عامل‌های پایین‌دستی، بعد از دریافت پیغام، با استفاده از متد receiveMessage، از عامل بالادستی خود درخواست داده‌ی به‌روزشده را می‌کنند. این درخواست در سلسله‌مراتب بالا می‌رود و در نهایت با فراخوانی متد getData به عامل ریشه‌ای می‌رسد. سپس داده دریافت‌شده و سلسه‌مراتب را پایین می‌رود تا به درخواست‌کننده برسد.
9. هر یک از عامل‌های پایین‌دستی سپس با استفاده از متد draw و با استفاده از داده‌ی تهیه‌شده، خود را آپدیت می‌کند.
10. حالا حالتی را تصور کنیم که observer ما یک View Handler باشد و داده‌ی آپدیت‌شده، بیانگر نیاز به حذف یکی از ویوها باشد. این View Handler با فراخوانی متد getData داده‌ی جدید را دریافت کرده و سپس با فراخوانی متد

[22] cache memory

update خود متوجه نیاز به بسته‌شدن یک View می‌شود. سپس با فراخوانی متد close برای آن view، آن را حذف می‌کند.

دقت کنیم تغییر داده و دسترسی در این تعاملات، منجر به تغییر آنی و در View ها می‌شود.



aDataSupplierProxy
aDataSupplier
aConcreteView1
aConcreteView2
aConcreteView2
top level agent control
aConcreteView2
second level agent
aConcreteView2
last level agent
aViewHandler
setData
setData
setCacheData
notify
updateData
getData
getData
if cache hit: return data
if cache miss: get data from
provider and update cache
Update other registered
ChangeObserver instances
updateData
getData
getData
update
sendMessage(change)
receiveMessage(change)
receiveMessage(change)
getData
receiveMessage(getData)
receiveMessage(getData)
draw
updateData
getData
getData
update
close
close

معایب این معماری

ایراداتی وارد به این معماری به نظر می‌رسد که اجمالا به صحبت در مورد آنها می‌پردازیم:

1. سربار تعاملات[23] در این سیستم توزیع‌شده بسیار بالاست، هر یک از تعاملات بایستی از چند مرحله indirection عبور کرده و به مقصد برسد. نرخ بالای message passing می‌تواند بالقوه تاثیر زیادی روی کارایی[24] سیستم بگذارد.
2. استفاده از الگوی PAC برای نمایش رابط کاربری می‌تواند بالقوه باعث کاهش چشمگیر کارایی و سرعت‌پاسخگویی[25] سیستم شود. این الگو امروزه به ندرت مورد استفاده قرار می‌گیرد.
3. استفاده از موجودیت‌های کارگردان در سیستم باعث کاهش چشمگیر پیچیدگی در محاسبات و تعاملات می‌شود اما معایب قابل توجهی نیز دارد.
   a. سربار تعاملات ناشی از ایجاد indirection و حجم بالای message passing می‌تواند باعث کاهش کارایی سیستم شود.
   b. این کلاس‌ها کارگردان هستند و نباید خود انجام‌دهنده‌ی محاسبات سیستمی باشند. این نباید ممکن است به اشتباه در طول عمر نرم‌افزار محقق شده و این کلاس‌ها را تبدیل به کلاس‌هایی چندوجهی، بزرگ و با تغییرات منتشرشونده و بی‌دلیل کند (به اصطلاح God Class).
   c. این کلاس‌ها در مرکز تعاملات و محاسبات قرار می‌گیرند و می‌توانند تبدیل به یک single point of failure برای سیستم شوند.

[23] communication overhead

[24] performance

[25] response time

مساله دوم

در این بخش از تمرین، دو idiom از الگوهای مطرح شده در کتاب Effective Java را در قالب محیط-مساله-راهکار مورد بررسی قرار می‌دهیم.

Use varargs judiciously

محیط

در زبان جاوا،گاهی نیاز است توابعی داشته باشیم که تعداد متغیری آرگومان[26] دریافت می‌کنند. به عنوان مثال تابعی را در نظر بگیرید که تعدادی عدد صحیح دریافت کرده و حاصل جمع آنها را محاسبه می‌کند. برای این مقصود از امکانی در جاوا به نام varargs استفاده می‌کنیم. این ویژگی به ما این آزادی را می‌دهد که آرگومان‌های ورودی تابع را به صورت یک لیست در اختیار داشته باشیم. به عنوان مثال:

```
public static int sum(int... args) {
     int result = 0;
     for (int arg: args) {
          result += arg;
     }
     return result;
}
```

مساله

حالتی را در نظر بگیرید که نیاز داشته باشیم تابعی ۱ یا بیشتر آرگومان ورودی بپذیرد. مثلا تابعی را در نظر بگیرید که تعدادی عدد صحیح را دریافت کرده و بیشترین آنها را محاسبه می‌کند. یک راه رسیدن به این هدف با استفاده از varargs، چک کردن تعداد آرگومان‌های ورودی در زمان اجرا و تخصیص خطا (Exception) مناسب در صورت خالی بودن لیست آرگومان‌هاست.

[26] argument

```java
public static int max(int... args) {
     if (args.length == 0) {
          throw new IllegalArgumentException("At least one
argument is required.");
     }
     int max = args[0];
     for (int i = 1; i < args.length; i+=1) {
          if (args[i] > max) {
              max = args[i];
          }
     }
     return max;
}
```

روش دیگری که به ذهن می‌رسد، مقدار دهی اولیه‌ی max با Integer.MAX_VALUE است:

```java
public static int max(int... args) {
     int max = Integer.MAX_VALUE;
     for (int arg: args) {
          if (arg > max) {
              max = arg;
          }
     }
     return max;
}
```

مقدار دهی اولیه max نازیباست و از طرفی این تابع برای حالتی که ورودی وجود نداشته باشد نیز جواب برمی‌گرداند.

راهکار

تابعی با دو ورودی، یکی ورودی عادی و دیگری varargs تعریف کنیم.

```
public static int max(int first, int... rest) {
     int max = first;
     for (int arg: rest) {
          if (arg > max) {
              max = arg;
          }
     }
     return max;
}
```

زیبایی این روش در این است که چک تعداد ورودی‌ها در زمان compile انجام می‌شود.

مورد تامل

استفاده از varargs باعث می‌شود یک لیست در حافظه‌ی heap به ازای هر فراخوانی ایجاد شود. از این رو در محیطهایی که حافظه محدود است، استفاده از این روش شاید ایده‌ی مناسبی نباشد. البته در شرایطی که می‌دانیم بخش عمده‌ای از فراخوانی‌ها با تعداد محدودی آرگومان صورت می‌پذیرد، (مثلا ۹۸ درصد با حداکثر ۳) می‌توانیم با overload کردن تابع با تعداد مختلف ورودی هم از امکان varargs استفاده کنیم و هم تا حد ممکن جلوی بروز مشکلات ناشی از کمبود منابع را بگیریم. به عنوان مثال:

```
static int mx(int first)
static int mx(int first, int second)
static int mx(int first, int second, int third)
static int mx(int first, int second, int third, int... rest)
```

Return empty collections or arrays not nulls

محیط

معمولا در برنامه‌نویسی سیستمی با متودهایی مواجه می‌شویم که باید آرایه یا کلکسیونی (مثل لیست و مجموعه) از عناصر را برگردانند. این آرایه یا کلکسیون می‌تواند خالی هم باشد.

مساله

معمولا در زبان جاوا عجیب نیست که متودی را ببینیم که که کلکسیون یا آرایه‌ای از عناصر را برمی‌گرداند یا در صورت خالی بودن این کلکسیون یا آرایه null برمی‌گرداند. چنین رویکردی در قبال نوشتار این برنامه نه‌تنها باعث پیچیده‌تر شدن برنامه شده (چک کردن حالت خاص و برگرداندن null) بلکه کد تمام clientهایی که از این برنامه استفاده خواهندکرد را نیز پیچیده می‌کند. چرا که این کلاینت‌ها بایستی حالت خاص برگرداندن null را نیز در کد خود چک کنند و عمل متناسب را انجام دهند. چنین کاری احتمال بروز خطا را نیز افزایش می‌دهد. زیرا برنامه‌نویس کلاینت ممکن است چک کردن حالت خاص را فراموش کند و این حالت خاص هم مثلا در production تا مدت‌ها رخ ندهد.

```
private final List<Student> registeredStudents = ...;

public List<Student> getRegisteredStudents() {
    if (registeredStudents.isEmpty()) {
        return null;
    }
    return new ArrayList<>(registeredStudents);
}
```

راهکار

به‌جای آنکه در حالتی که کلکسیون یا آرایه خالی است، null برگردانیم، یک کلکسیون یا آرایه‌ی خالی برگردانیم. در این صورت نیاز به چک حالت خاص نه سمت سرور و نه سمت کلاینت وجود نخواهد داشت. همینطور احتمال بروز خطا نیز کاهش پیدا کرده و کد نیز قابل‌درک تر و باکیفیت‌تر خواهد شد.

```java
private final List<Student> registeredStudents = ...;

public List<Student> getRegisteredStudents() {
    return new ArrayList<>(registeredStudents);
}
```

مورد تامل

در شرایط بسیار خاص کمبود شدید منابع پردازشی ممکن است بیان شود ساخت یک کلکسیون یا آرایه جدید برای هربار فراخوانی این متد به صرفه نیست. در چنین شرایطی می‌توان از عناصر immutable ای مانند حاصل‌های فراخوانی توابع emptyList، emptySet و emptyMap از Collections بهره برد یا برای حالتی که خروجی باید یک آرایه باشد، یک آرایه خالی ساخت و همیشه آن را برگرداند (آرایه‌ی خالی در جاوا immutable است).

```java
private final List<Student> registeredStudents = ...;

public List<Student> getRegisteredStudents() {
    if (registeredStudents.isEmpty()) {
        return Collections.emptyList();
    }
    return new ArrayList<>(registeredStudents);
}
```

منابع

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "**Design Patterns: Elements of Reusable Object-oriented Software.**" *Addison-Wesley, 1995.*

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., "**Pattern-Oriented Software Architecture: A System of Patterns, Vol. 1.**" *Wiley, 1996.*

[3] Ramsin, R. **"Patterns in Software Engineering graduate course"** lecture notes. Lectures 2-9. *Department of Computer Engineering, Sharif University of Technology, 2022.*

[4] Joshua Bloch, "**Effective Java**" 3rd edition, ISBN: 0134685997, *Addison-Wesley Professional; (December 27, 2017).*