Od dveh generalov do dveh storitev

Matjaž Prtenjak

Endava, d. o. o., Ljubljana, Slovenija
matjaz.prtenjak@endava.com

V svetu umetne inteligence se hitro ujamemo v zanko, da zaradi njene pomoči razvijamo preveč kompleksne sisteme. V tem duhu in duhu spleta ter neomejenih možnosti tako zagrabimo tudi idejo mikrostoritev. Sistem razdelimo na manjše storitve, ki imajo praviloma lastništvo nad svojimi podatki, komunikacija med njimi pa poteka prek omrežja, oddaljenih klicev ali sporočil. Rešitev je videti dobra, naravna in vsi jo priporočajo, pri tem pa zamolčijo kompleksnost. Klic funkcije v dinamični knjižnici zamenjamo s komunikacijo po nezanesljivem omrežju. Lokalno transakcijo nadomestimo s porazdeljeno, preprost vrstni red izvajanja pa z dogodki, ki lahko pridejo večkrat, nikoli ali pa celo v povsem napačnem vrstnem redu. Izkaže se, da mikrostoritve niso tehnična rešitev, temveč organizacijska rešitev za dovolj velike organizacije.

Ključne besede:
mikrostoritve
distribuirani sistemi
modularni monolit
sporočilni sistemi
kompenzacijska dejanja

Univerzitetna založba Univerze v Mariboru

ISBN 978-961-299-188-3
DOI https://doi.org/10.18690/um.feri.7.2026.7

1 Uvod – dva generala

Dva generala sta osnovna premisa članka, ki ponazarja ključne težave distribuirane arhitekture.

S svojima vojskama sta na nasprotnih straneh mesta in lahko zmagata samo, če napadeta usklajeno. Komunicirata lahko le s sli, ki pa jih lahko sovražnik prestreže (oziroma katerih sporočila se lahko izgubijo).

- Prvi general pošlje sporočilo: "Napadli bomo opolnoči." – vendar ne ve, ali je drugi general prejel sporočilo, zato čaka na potrditev.
- Drugi pošlje odgovor: "Dogovorjeno." – vendar ne ve, ali je prvi general prejel potrditev, zato čaka na potrditev potrditve.
- Prvi general pošlje odgovor na odgovor: "Prejel potrdilo!", vendar spet ne ve, ali je drugi general prejel potrditev potrditve, zato čaka.
- Drugi general pošlje sporočilo: ...

Veriga potrditev se lahko nadaljuje v neskončnost, popolne gotovosti pa ne dosežemo. Problem koordiniranega napada je bil formalno obravnavan že v zgodnjih delih o komunikacijskih protokolih [1].

Lahko pa zadevo še bolj zapletemo.

Problem »bizantinskih generalov« [2] opisuje še zahtevnejši scenarij, v katerem lahko posamezen udeleženec ali komponenta deluje napačno oziroma zlonamerno in različnim udeležencem pošilja nasprotujoče si informacije. Tu ne govorimo več samo o nezanesljivosti dostave, temveč tudi o nezanesljivosti vsebine in vedenja udeležencev.

2 Ko se preprosti klic spremeni v dogovarjanje

Pri monolitni aplikaciji je lahko veliko stvari presenetljivo dolgočasnih. Metoda pokliče drugo metodo, knjižnica vrne rezultat, transakcija se potrdi ali zavrne, izjema pa se običajno pojavi v istem procesu in v istem programu.

To seveda ne pomeni, da je monolitna aplikacija privzeto enostavna. Kot vemo, se namreč navadno izkaže prav nasprotno. Monolitne aplikacije so lahko presenetljivo kompleksne in zapletene. Mnogokrat postanejo skupek programske kode, ki je medsebojno tako prepletena, da se je nihče ne upa več dotakniti [3].

Imajo pa monolitne aplikacije eno zelo pomembno lastnost: **vsa komunikacija poteka lokalno.** Če modul A pokliče modul B, ne razmišljamo o:

- izgubljenem paketu,
- ponovnem pošiljanju sporočila,
- potrditvi prejemnika,
- tem, ali je prejemnik procesiral zahtevo, preden je padla povezava.

Mikrostoritve pa vse te predpostavke obrnejo na glavo:

Monolitna aplikacija	**Mikrostoritve**
Klic funkcije je lokalen in neposreden	Komunikacija poteka prek HTTP-ja, RPC-ja ali sporočil
Ena lokalna transakcija	Več lokalnih transakcij in usklajevanje med storitvami
Skupna podatkovna baza (lahko celo del istega procesa!)	Storitve imajo praviloma ločeno lastništvo nad podatki
Napake in ponovitve so običajno lažje obvladljive	Sporočila so lahko podvojena, zakasnjena, preurejena ali izgubljena, odvisno od jamstev sistema

Mikrostoritve so distribuirani sistemi z vsemi težavami, ki jih slednji prinašajo. Mikrostoritve niso zahtevne zato, ker uporabljamo zveneča tehnična imena, kot so SAGA, Inbox, Outbox ali pa Eventual Consistency. Mikrostoritve so zahtevne zato, ker je **nemogoče popolnoma odstraniti negotovost, zakasnitev, podvajanje in delne odpovedi.**

3 Od vsakdanjih metafor do tehničnih problemov

- *Potujete … Hotel, letalo in avtomobil ste rezervirali neodvisno na treh spletnih straneh. Na letališču dobite sporočilo, da je rezervacija hotela zavrnjena. Kaj storiti? Prekiniti potovanje, potovati in upati, da najdete drugi hotel?*

Če rezerviramo hotel, letalo in avtomobil na treh ločenih spletnih mestih, pričakujemo eno potovanje, sistemsko pa imamo tri ločene poslovne transakcije. Če rezervacija hotela odpove po tem, ko sta letalo in avtomobil že potrjena, ni več preproste možnosti "rollback". Sistem lahko prekliče let, odpove avtomobil ali ponudi drug hotel, vendar je to poslovna odločitev, ne samo tehnična operacija.

To je problem distribuiranih transakcij.

- *Kolesarite in imate hudo nesrečo. Vzamete mobilni telefon in pokličete pomoč… oglasi se operater in začnete govoriti, kje ste, kaj se je zgodilo …, nato pa vidite, da se je telefon vmes ugasnil. Kaj storiti? Ali je operater razumel, da ste imeli nesrečo? Kdaj je telefon nehal delovati?*

Morda je operater razumel lokacijo in naravo nesreče, morda samo prvo polovico stavka, morda ničesar.

V mikrostoritvah se to pokaže kot delno izvedeno dejanje: ena storitev je nekaj zapisala, druga je dobila samo del podatkov, tretja pa se je odzvala prepozno. Če aplikacija tak dogodek samo ponovi, lahko stvar popravi ali pa naredi dodatno škodo. Zato morajo biti ponovitve načrtovane, operacije pa po možnosti idempotentne.

- *Nujno morate na sestanek iz Maribora v Ljubljano. Na avtocesti so hudi zastoji. Kaj storiti? Poskusiti vseeno na avtocesti ali iti po stari cesti?*

Če stojimo v zastoju med Mariborom in Ljubljano, izbiramo med slabo in nepopolno informacijo: ostati na avtocesti ali iti po stari cesti.

Tudi mikrostoritve živijo v okolju, kjer je omrežje včasih počasno, posrednik sporočil preobremenjen ali pa podatkovna baza ne deluje.

Vzorec ponovnih poskusov (*»retry«*) poudarja, da so prehodne napake v oblačnih in distribuiranih okoljih normalne ter da mora aplikacija odpovedi obravnavati z ustrezno zakasnitvijo in omejenim številom poskusov. Vendar *ponovni poskus* ni univerzalno zdravilo. Če vse storitve hkrati agresivno ponavljajo zahteve, lahko iz prehodne napake naredijo trajnejšo preobremenitev.

- *Od stranke prejmete naročilo po elektronski pošti in naročilo za iste izdelke tudi po navadni pošti. Ali je to isto naročilo ali moram dvakrat dostaviti iste izdelke?*

Če stranka pošlje naročilo po elektronski in navadni pošti, je poslovno vprašanje, ali gre za isto naročilo ali za dve naročili.

Tehnično podobno vprašanje se pojavi pri sporočilih: posrednik sporočil lahko iz različnih razlogov dostavi isto sporočilo več kot enkrat. Tudi če je posrednik zanesljiv, se lahko potrošnik sesuje po obdelavi, vendar pred potrditvijo.

Po ponovnem zagonu dobi isto sporočilo znova. Rešitev ni slepo zaupanje infrastrukturi, temveč deduplikacija na poslovni ravni: enolični identifikatorji.

- *Na telefon dobite sporočilo DHL-a: „Paket dostavljen". Greste pred hišo in iščete, kje je kurir pustil paket, nakar dobite novo sporočilo: „Paket prevažamo". Katero sporočilo je pravilno? Ali je paket vseeno dostavljen, pa ga je nekdo ukradel?*

Če prejmemo najprej obvestilo "paket dostavljen", čez minuto pa "paket prevažamo", imamo problem vrstnega reda dogodkov.

V centraliziranem sistemu je vrstni red pogosto samoumeven: ena transakcija se zgodi pred drugo. V distribuiranem sistemu pa lahko dogodek potuje skozi vrsto, ponovno dostavo, transformacijo, shrambo in potrošnika, ki deluje z drugačno hitrostjo.

Zato se moramo vprašati, ali dogodki vsebujejo različico, zaporedno številko ali časovno oznako, in ali zna prejemnik ignorirati starejši dogodek.

- *Greste na upravno enoto, da bi spremenili naslov…*

Če na upravni enoti spremenimo stalni naslov, pa zdravstveni dom še nekaj časa pošilja pošto na stari naslov, doživimo neusklajenost podatkov. Včasih to ni katastrofa, včasih pa je nesprejemljivo.

Glavna arhitekturna odločitev ni, ali si neusklajenost želimo, temveč, kje jo lahko sprejmemo, kje jo moramo zmanjšati in kje je sploh ne smemo uporabiti.

4 Mikrostoritve so organizacijska, ne samo tehnična arhitektura

Mikrostoritve so postale priljubljene zato, ker jih uporabljajo velika podjetja. In če jih uporabljajo velika podjetja, potem morajo biti »dobre«! Toda to je nevarna poenostavitev.

Velika podjetja imajo veliko razvijalskih ekip, ločene domene, kompleksne postopke, različne tehnologije in potrebe, saj se deli organizacije premikajo neodvisno drug od drugega. Tam mikrostoritve pogosto rešujejo problem koordinacije med ljudmi. V manjšem sistemu pa lahko isti pristop ustvari več komunikacijskih težav, kot jih reši.

Conwayevo pravilo pravi, da organizacije oblikujejo sisteme, ki odražajo njihove komunikacijske strukture [4]. To je eden najpomembnejših razlogov, zakaj mikrostoritve pogosto delujejo v velikih organizacijah: meje med storitvami lahko sledijo mejam med enotami. To ni samo tehnični opis. Je tudi organizacijski opis.

Samostojna storitev ima največ smisla, če obstaja tudi samostojen lastnik: enota, ki razume domeno, razvija funkcionalnost, upravlja podatke, spremlja produkcijo in se odziva na incidente. Če organizacija teh sposobnosti nima, mikrostoritve ne bodo zmanjšale kaosa; samo razpršile ga bodo po omrežju.

To je tudi razlog, zakaj je trditev "mikrostoritve rešujejo skalabilnost" premalo natančna.

Da, posamezno storitev lahko skaliramo neodvisno.

Da, težko komponento lahko ločimo od manj obremenjenih delov.

Toda tudi monolit lahko horizontalno skaliramo, če le ni preslabo zasnovan, in lahko interno uporablja asinhrono obdelavo, predpomnjenje in delitev podatkovnih obremenitev. Mikrostoritve rešujejo specifične probleme skaliranja, ne pa vseh problemov skaliranja. Pogosto bolj kot strežniško skalabilnost rešujejo organizacijsko skalabilnost.

Zato je praktično vprašanje drugačno: ali je težava v tem, da se aplikacija tehnično ne more skalirati, ali v tem, da se ljudje ne morejo več uskladiti?

Če je glavni problem usklajevanje več ekip, dolgi cikli izdaj in nejasno lastništvo domen, so mikrostoritve lahko smiselne.

Če je problem predvsem neurejena koda, slabe meje modulov in manjkajoči testi, mikrostoritve ne bodo pomagale. Takrat bomo iz slabe monolitne aplikacije naredili še slabše mikrostoritve.

5 Saga: ko rollback ni več ena sama operacija

Klasična ACID transakcija [5] deluje dobro, dokler je meja transakcije dovolj majhna in pod nadzorom enega sistema. V mikrostoritvah pa ima vsaka storitev pogosto svojo bazo in svoj model podatkov.

Globalna transakcija prek več storitev je mogoča, vendar običajno poveča soodvisnost med storitvami, zahteve do infrastrukture in občutljivost sistema na nedosegljivost posameznih komponent. Zato se za številne daljše poslovne procese uporablja vzorec Saga [6]. Izvirni koncept sage je pravzaprav zaporedje transakcij, kjer se ob delni napaki izvedejo kompenzacijske transakcije. To so transakcije, ki popravijo ali izničijo predhodne korake.

V sodobnih mikrostoritvah saga pomeni nekaj podobnega: namesto ene velike transakcije izvedemo več lokalnih transakcij. Vsak korak potrdi svoje spremembe, ob napaki pa sistem izvede poslovno smiselne kompenzacije.

Pri monolitni aplikaciji lahko takšen proces pogosto izvedemo v eni transakciji ACID, zato velja načelo »vse ali nič«. V realnem svetu pa je lahko odpoved leta dražja kot iskanje drugega hotela. Zato kompenzacija ni vedno matematični "undo". Je poslovni postopek. Če ni hotela, lahko ponudimo drug hotel, delni dobropis, človeško intervencijo ali odpoved celotnega paketa. Vzorec kompenzacijskih transakcij (»*compensating transaction*«) opozarja, da kompenzacija v distribuiranih sistemih ni nujno preprosto vračanje korakov v obratnem vrstnem redu, temveč zahteva poslovna pravila.

Vzorec saga poznamo v dveh oblikah: orkestrirana saga in koreografirana saga. Pri orkestrirani sagi obstaja centralni orkestrator. Ta pozna tok procesa, pošilja ukaze posameznim storitvam, spremlja stanje in ob napaki sproži kompenzacije. Prednost je preglednost. Lažje je videti, kje je proces, lažje je dodati nadzor in lažje je razložiti poslovni tok. Slabost je, da orkestrator postane dodatna komponenta, ki jo moramo razvijati, testirati, skalirati in opazovati. Če je slabo zasnovan, lahko postane enotna točka odpovedi (»*single point of failure*«) ali centralna poslovna logika, ki zmanjšuje avtonomijo storitev.

Pri koreografirani sagi centralnega dirigenta ni. Storitve objavljajo dogodke, druge storitve jih poslušajo in nadaljujejo svoje korake. Tak pristop je bolj naraven v dogodkovno vodenih sistemih (»*event-driven systems*«) in ohranja več avtonomije. Vendar ima drugačno ceno: tok procesa je razpršen po več storitvah. Hitro lahko pridemo do množice dogodkov, ki jim sploh ne moremo več slediti in se jih bojimo dotikati. Iz slabe programske kode v monolitni aplikaciji pridemo do nepregledne množice dogodkov, ki jih več ne obvladamo, ker nihče več ne ve, kaj je sprožilo kaj in kateri dogodek je še vedno relevanten... in spet smo pri slabi aplikaciji!

Praktično pravilo je naslednje: če je poslovni proces dolg, kritičen, ima jasen življenjski cikel in ga je treba nadzorovati, je orkestracija pogosto razumljivejša.

Če so dogodki naravni del domene, so koraki šibko sklopljeni in ni enega jasnega lastnika procesa, je koreografija lahko primerna.

V obeh primerih pa saga ni čarobna zamenjava za transakcije. Je eksplicitna odločitev, da konsistentnost dosežemo s procesom, opazovanjem, idempotentnostjo in kompenzacijo.

6 Outbox, inbox in dve hudi težavi: *zombie record in ghost message*

Storitev mora pogosto narediti dve stvari: spremeniti podatke v svoji bazi in objaviti sporočilo v sporočilno vrsto.

Če najprej zapiše v bazo in nato pade pred objavo sporočila, dobimo zapis, za katerega drugi deli sistema nikoli ne izvedo. Takšnim zapisom pravimo »*zombie record*«: zapis obstaja, vendar proces, ki bi moral slediti, ni zaživel.

Če pa storitev najprej objavi sporočilo in nato pade pred potrditvijo transakcije v bazi, lahko drugi sistemi reagirajo na dogodek, ki ga baza ne vsebuje. Temu lahko rečemo »*ghost message*«: sporočilo obstaja, poslovno dejstvo pa ne.

Vzorec Outbox rešuje prav ta razkorak. Storitev v isti lokalni transakciji zapiše poslovno spremembo in vrstico v tabelo Outbox. Ločen proces nato bere Outbox in sporočila objavlja v sporočilno vrsto.

Vzorec Outbox ne zagotavlja samodejno dostave ali obdelave »točno enkrat«. Če proces za objavljanje pade po uspešni objavi v posredniku, vendar pred označitvijo zapisa kot obdelanega, se lahko sporočilo objavi večkrat. Zato mora potrošniška storitev uporabljati mehanizem Inbox.

Vzorec Inbox je torej druga polovica zgodbe. Prejemnik mora vedeti, ali je sporočilo z določenim identifikatorjem že obdelal. Eden od praktičnih pristopov je tabela ProcessedMessages z enoličnim ključem, na primer kombinacijo SubscriberId in MessageId. Potrošnik ob začetku transakcije vstavi ID sporočila v tabelo obdelanih sporočil; če vnos zaradi primarnega ključa ne uspe, je bilo sporočilo že obdelano in ga lahko ignorira.

Vendar je pri tem ključna razlika med različnimi identifikatorji! MessageId je identiteta konkretnega tehničnega sporočila. BusinessId je identiteta poslovnega dogodka ali entitete. Poznamo pa jih seveda še več: *IssueId, ReservationId, PaymentId, OrderNumber* in podobni.

Če sistem generira nov identifikator pri vsakem ponovnem pošiljanju, deduplikacija po tem identifikatorju ne bo ujela poslovnega dvojnika. Zato mora arhitektura jasno določiti, kateri identifikator rešuje katero težavo. Vem, sliši se skrajno preprosto, a se v praksi izkaže, da temu ni ravno tako!

MessageId pomaga pri tehnični ponovni dostavi istega sporočila. BusinessId pomaga pri vprašanju, ali gre za isto poslovno naročilo, isto prijavo okvare ali isto rezervacijo.

7 Retry, timeout in circuit breaker: varovalke namesto panike

Napake v distribuiranem sistemu niso izjema, ampak del običajnega delovanja.

Povezava se prekine. Storitev potrebuje več časa. Sporočilna vrsta je začasno nedosegljiva. Podatkovna baza naredi »*failover*«. Zato so časovne omejitve (»*timeout*«) in ponovni poskusi osnovna higiena. Brez časovnih omejitev lahko nit čaka predolgo in porabi vire, ki jih potrebujejo drugi deli sistema. Brez ponovnih poskusov pa lahko kratka prehodna napaka povzroči nepotreben poslovni neuspeh.

Toda ponovni poskus mora biti varen. Če ponavljamo branje podatkov, je tveganje majhno. Če ponavljamo plačilo, rezervacijo ali pošiljanje elektronske pošte, pa je to nevarno. Pred tem moramo zagotoviti idempotentnost.

Ponovni poskusi z naraščajočim časovnim zamikom in naključnim odmikom (»*backoff, jitter*«) zmanjšajo možnost, da bi vse instance sistema ponavljale zahteve v istem ritmu. Ponovne poskuse uporabljajmo predvsem pri prehodnih napakah. Pri trajnih napakah ali neveljavnih zahtevah ponavljanje običajno nima smisla.

Tu moramo namreč biti pazljivi. Predstavljajmo si veliko odjemalcev, ki poskušajo doseči strežnik, ki pa ima določene težave. In ravno, ko se strežnik »pobere«, ga spet zasujejo vsi odjemalci, zato strežnik spet »počepne«. Ponovni poskusi morajo torej imeti naključno dolg vmesni čas čakanja in ta čas naj postopoma raste.

Odklopnik (»*circuit breaker*«) rešuje drugo plat tega problema. Če odvisna storitev očitno ne deluje, nima smisla, da jo vsaka zahteva znova obremenjuje. Odklopnik po določenem pragu napak zadevo prekine in klice začasno zavrne ali preusmeri.

Kasneje v napol odprtem stanju (»*half-open*«) dovoli omejeno število poskusov, da preveri, ali se je storitev popravila. Odklopnik preprečuje ponavljajoče izvajanje operacije, ki bo verjetno neuspešna, in s tem pomaga preprečiti kaskadne napake.

Pregledi zdravja storitve (»*health checks*«) so naslednji del te zgodbe. Storitev mora znati povedati, ali deluje in ali je pripravljena sprejemati promet. Končna točka za preverjanje zdravja ni samo privlačen naslov URL »/health«. Je dogovor med aplikacijo, orkestratorjem, uravnalnikom obremenitev (»*load balancer*«) in nadzornikom.

Vse omenjene tehnike – časovna omejitev, ponovni poskus, odklopnik in preverjanje zdravja – morajo biti usklajene. Kratke časovne omejitve brez ponovnih poskusov lahko povzročijo preveč napak. Prepogosti ponovni poskusi brez naključnih zakasnitev pa lahko povzročijo kaskadno odpoved sistemov zaradi prevelikega prometa.

Zato jih ne smemo obravnavati kot knjižnične nastavitve, temveč kot del arhitekturnega dogovora: katere operacije se smejo ponoviti, koliko časa smejo trajati, kdaj odpovemo hitro in kako operaterji vidijo stanje sistema.

8 Vrstni red, verzije, stanja in eventualna konsistentnost

Dogodki v mikrostoritvah pogosto ne pridejo v vrstnem redu, ki ga pričakujemo. Vzrok je lahko sporočilna vrsta, več pošiljateljev, paralelizem, ponovna dostava, različne poti dogodkov ali preprosto različen čas obdelave. Če prejemnik nekritično sprejme vsak dogodek, lahko starejši dogodek prepiše novejše stanje. Zato mora dogodek nositi dovolj informacij, da prejemnik sprejme pravilno odločitev.

Najpogostejši mehanizmi so verzije, zaporedne številke in grafi stanja (»*state machine*«). Če ima entiteta verzijo, lahko potrošnik dogodek verzije X ignorira, če je že obdelal dogodek verzije X+1.

Če ima tok dogodkov zaporedno številko, lahko zazna luknjo in sproži ponovno sinhronizacijo. Če ima proces eksplicitna stanja, se lahko sistem odloči, da prehod iz "Dostavljeno" nazaj v "V prevozu" ni veljaven, razen če gre za poseben poslovni scenarij.

Konsistentnost dogodkov je pri tem pogosto nujna, vendar mora biti vidna. Slabo je, če uporabnik misli, da je sprememba naslova takoj veljavna v vseh sistemih, sistem pa v resnici potrebuje minute ali ure, da sinhronizira podatkovne baze. Bolje je, da domena jasno pove, kaj je vir resnice in kdaj so drugi pogledi posodobljeni.

Pri usklajevanju stanj so koristna tudi periodična opravila usklajevanja (reconciliation jobs). Ta primerjajo stanje med sistemi, odkrijejo odstopanja in jih usmerijo v samodejno ali ročno popravljanje. Če izhodno sporočilo ni bilo objavljeno, ga lahko opravilo pozneje ponovno objavi. Če potrošnik dogodka ni obdelal, se lahko obdelava ponovi. Če se stanje dveh sistemov še vedno razlikuje, opravilo odstopanje evidentira in sproži ustrezen popravljalni postopek.

Pomembno je, da takšna opravila niso priznanje poraza. So priznanje, da distribuirani sistem ne more temeljiti na popolni gotovosti!

9 Single point of failure: mikrostoritve ne odpravijo centralnih odvisnosti

Pogosta iluzija je, da mikrostoritve samodejno pomenijo večjo odpornost. V resnici lahko hitro ustvarimo sistem, kjer je vse razdeljeno, razen ene kritične točke. Lahko imamo deset storitev, vse pa potrebujejo isto sporočilno vrsto ali pa istega SAGA koordinatorja. Lahko imamo ločene baze, vse pa uporabljajo isti DNS ali isto Kubernetes gručo...

Zato je treba enotno točko odpovedi obravnavati širše kot samo "ena baza" ali "ena sporočilna vrsta". Gre za vsako komponento, brez katere sistem ne more opravljati ključne poslovne funkcije. Včasih je rešitev redundanca: gruče, replike, visoka razpoložljivost, rezervna pot. Včasih je rešitev degradacija: če priporočilni sistem ne deluje, lahko osnovna funkcionalnost vseeno deluje. Včasih je rešitev izolacija: napaka v poročilih ne sme ustaviti obdelave kritičnih prijav.

Zanesljiva izmenjava sporočil zahteva več kot samo njihovo shranjevanje. Pomembno je razlikovati med tem, ali sporočilo preživi ponovni zagon sistema, ali je posrednik trenutno dosegljiv in kako sistem razume uspešno obdelavo sporočila. Obstojna sporočila, trajne čakalne vrste ter potrditve pošiljanja in prejema zmanjšujejo tveganje izgube podatkov.

Kljub temu mora sistem ob nedosegljivosti posrednika določiti, ali bo sporočilo shranil lokalno v odhodni predal in ga poslal pozneje, ali pa bo uporabniku vrnil napako. Prav tako mora določiti, kaj se zgodi, če storitev, ki naj bi sporočilo obdelala, ne deluje: sporočilo lahko nekaj časa čaka, ali pa se sproži eskalacijski postopek.

Orkestrirana saga lahko postane ranljiva, če orkestrator izgubi stanje in po ponovnem zagonu ne more ugotoviti, kateri koraki so bili že izvedeni. Pomembno je tudi, da ponovna izvedba istega ukaza ne povzroči neželenih učinkov. Kljub temu koreografija ni nujno varnejša, saj lahko pomembno odvisnost skrije drugje, na primer v posredniku sporočil, skupnih pravilih za dogodke ali znanju posamezne ekipe.

Pri izbiri arhitekture zato ni najpomembnejše vprašanje, ali imamo centralno ali decentralizirano rešitev. Pomembneje je razumeti, kje so odvisnosti, kdo jih vzdržuje in kaj se zgodi, ko odpovejo.

10 Predlog: ne zapletajte po nepotrebnem

Mikrostoritve so uporabne, kadar rešujejo resničen problem avtonomije, lastništva in neodvisnega uvajanja. Če imamo tri razvijalske ekipe, to še ne pomeni, da mora vsaka ekipa dobiti svojo mikrostoritev, svojo bazo, svoje sporočilne pogodbe, svoje incidente in svojo omrežno zakasnitev.

Pogosto je boljši prvi korak modularni monolit. To je sistem z jasno ločenimi moduli, notranjimi mejami, ločenimi imenskimi prostori, jasnim lastništvom kode in po možnosti ločenimi shemami ali vsaj strogo nadzorovanimi dostopi do podatkov. Moduli lahko komunicirajo prek dobro definiranih vmesnikov ali notranjih dogodkov, vendar ostanejo v istem procesu. Če se kasneje izkaže, da mora nek modul živeti samostojno, ga lažje izločimo, ker so meje že narisane.

Ali so NuGet/npm/pip/maven/... paketi mikrostoritve? Seveda niso. Vendar nas prisilijo v razmislek.

Knjižnica je lahko samostojna komponenta, ima lastnika, verzije, javni API in testni paket. Razlika je, da jo kličemo lokalno in pošljemo skupaj z aplikacijo. Če je cilj samo razdeliti odgovornost med razvojne ekipe, je knjižnica pogosto dovolj. Če je cilj ločeno skaliranje, ločeno uvajanje, ločen tehnološki sklad ali različna operativna odgovornost, je storitev bolj smiselna.

Storitve so komponente zunaj procesa in oddaljeni klici so dražji kot lokalni klici, zato morajo biti vmesniki bolj grobozrnati. To je praktičen kriterij. Če bodo nove mikrostoritve samo nadomestile vsak lokalni klic s klicem REST, bomo dobili počasnejši in bolj krhek sistem. Če pa storitev predstavlja jasno poslovno zmožnost, ima svoje podatke, svoj življenjski cikel in svojo ekipo, je komunikacijski strošek lahko upravičen.

11 Zaključek: arhitektura kot odločitev o stroških

Mikrostoritve same po sebi niso ne dobre ne slabe; njihova uvedba je predvsem odločitev o tem, katere kompromise je organizacija pripravljena sprejeti.

Pri monolitni arhitekturi moramo sprejeti skupno uvajanje celotne aplikacije, večji obseg kode in nevarnost, da notranje meje med funkcionalnostmi niso ustrezno določene.

Pri mikrostoritvah se ti izzivi prenesejo na omrežno komunikacijo, sporočilne sisteme, sčasoma doseženo konsistentnost podatkov ter zahtevnejše upravljanje in odpravljanje napak. Delovanje sistema je zato tudi manj predvidljivo.

Modularni monolit poskuša začasno združiti prednosti obeh svetov: jasne meje brez distribuirane kompleksnosti.

Pri uporabi mikrostoritev je smiselno upoštevati naslednja pravila:

1. Mikrostoritev ne izbirajmo zgolj zato, ker jih uporabljajo velika podjetja.
2. Aplikacije ne razdelimo na posamezne storitve, dokler ne razumemo domenskih meja in odgovornosti.
3. Poslovno pomembnih dogodkov ne objavljajmo brez vzorca Outbox oziroma zanesljivega lokalnega shranjevanja odhodnih sporočil.
4. Pri obdelavi sporočil zagotovimo idempotentnost, da ponovna obdelava ne povzroči neželenih učinkov.
5. Ponovnih poskusov ne izvajajmo brez časovnih omejitev, omejenega števila poskusov in ustreznega zamika med njimi.
6. Sage ne uvajajmo, dokler ne razumemo kompenzacijskih dejanj, potrebnih ob neuspehu posameznega koraka.
7. Zavedajte se, da bo prišlo do neusklajenosti podatkov in to sprejmite ter ustrezno ukrepajte že vnaprej.

Najboljša arhitektura ni tista z največ vzorci. Najboljša arhitektura je tista, pri kateri so vzorci uporabljeni tam, kjer rešujejo konkreten problem. Včasih je to saga. Včasih je to Outbox. Včasih je to samo dobro oblikovana knjižnica v istem procesu.

Arhitektura ni tekmovanje v številu škatlic na diagramu!

Literatura

[1] Two Generals' problem. Wikipedia. obiskano 8. 7. 2026, https://en.wikipedia.org/wiki/Two_Generals%27_Problem

[2] Byzantine fault, Wikipedia. obiskano 8. 7. 2026, https://en.wikipedia.org/wiki/Byzantine_fault

[3] Spaghetti code, Wikipedia. obiskano 8. 7. 2026, https://en.wikipedia.org/wiki/Spaghetti_code

[4] Conway, M. E. (1968). How do committees invent? Datamation, 14(4), 28-31. https://www.melconway.com/Home/Committees_Paper.html

[5] ACID, Wikipedia. Obiskano 08.07.2026, https://en.wikipedia.org/wiki/ACID

[6] Long-running transaction, Wikipedia. Obiskano 08.07.2026, https://en.wikipedia.org/wiki/Long-running_transaction