Rinnakkaisten ohjelmien testaus – Keskeisiä periaatteita ja strategioita

Matti Vuori, Tampereen teknillinen yliopisto

Sisällysluettelo

Kalvosarjan tarkoitus

- Kalvosarja tarkastelee rinnakkaisuuteen liittyvien asioiden testausta.
- Oletuksena on, että kuulija tuntee testauksen perusasiat ja tekniikat – niitä ei toisteta tässä sarjassa.
- Lähestymistapa on teknologia- ja testausvälineriippumaton ja käytännönläheinen.
- Tavoitteena on auttaa löytämään oikeita suhtautumistapoja, periaatteita ja strategioita, joilla kussakin testaustilanteessa osataan löytää oikeat testaustavat oikeille asioille, ymmärtäen testiympäristöjen rajoitukset.

Testauksen haasteet

- Rinnakkaisuuden ongelmat ovat arkipäivää.
  - Mystisiä bugeja – joskus toimii ja joskus ei.
  - Virheiden sijainti koodissa vaikea jäljittää.
  - Virheiden toistaminenkin on usein vaikeaa.
- Niiden testaus perinteisillä testausmenetelmillä on vaikeaa.
  - Perinteinen metodeja ja funktioita käsittelevä yksikkötestaus ei oikein toimi, välineet eivät tue tätä eivätkä löydä ongelmia.
    - Yksikkötestaus eristää asiat testausta varten, kun rinnakkaisuuden ongelmissa on kyse asioiden herkästä vuorovaikutuksesta.
    - Testityökalut, monitorointi ja logitus muuttavat ajoituksia.
  - Testiympäristöt eivät ole samanlaisia kuin tuotantoympäristöt – tuotantoympäristössä nousee esille uusia ongelmia.
  - Testausoppeja pitää siksi soveltaa luovasti.
- Jos jossain asiassa ”piru asuu yksityiskohdissa”, niin se pätee juuri rinnakkaisuusasioissa!

Keskeisiä periaatteita 1/3

- Ymmärrä, mihin käyttöön ohjelma ja komponentti joutuu.
  - Millaista käyttöä.
  - Millainen ajoympäristö.
  - Mitä kaikkia muita softia koneessa pyörii.
  - Mitä siellä tapahtuu.
  - Miten arkkitehtuuri toimii.
- Oleta, että jokaisessa komponentissa on virheitä.
  - Omassasi ja kaikissa muissa.
- Kuormitus nostaa viat esille.
  - Laita softa testeissä koville.
  - Varmista testaamalle, ettei se joudu koville käytössä (eli että sen ajoaikainen ympäristö ei testauksen perusteella vaadi liikoja).
- Odota pahinta kaikissa operaatioissa, jotka koskevat yhteisiä resursseja – ja testaa, että se pahinkin kestetään.

Keskeisiä periaatteita 2/3

- Oleta, että kaikki muut prosessit, säikeet jne... voivat tehdä mitä tahansa ja testaa, että se hallitaan ja siedetään.
- Kaikki voi muuttua päälaelleen, kun ajoympäristöä hieman muutetaan – eli muuta sitä testauksessa.
  - Odotettu suoritusnopeus.
  - Kuormitus.
  - Muiden komponenttien toiminta ja suorituskyky.
- Arvosta satunnaisuutta ja variaatiota testauksessa.
  - Muutoksilla ongelmat esille.
- Mitä tiukemmat ajoitusvaatimukset, sitä herkempi ohjelma on häiriöille.
- Liioittele testauksessa kaikkea – vielä hieman lisää...
- Testaa varautuminen tuleviin muutoksiin .
  - Jos vaikka prosessori vaihtuu – nopeampaan tai hitaampaan.
  - Uusi käyttöjärjestelmäversio on usein hitaampi.

Keskeisiä periaatteita 3/3

- Testattavien buildien jako kahtia:
  - Debug-versiot.
    - Voidaan kehityksen aikana logittaa ja muuten monitoroida mielin määrin ja yrittää ymmärtää toimintaa.
    - Vaarana keskittää testaus tähän versioon ja siten tehdä testausta, johon voi luottaa heikosti.
  - Release-konfiguraatio tärkein.
    - Vastaa sitä, mitä toimitetaan asiakkaalle.
    - Ei nopeutta (kenties) muuttavaa debuggauskoodia.
    - Koodin optimointioptiot lopulliset.
    - Matalan tason logitukset yms. pois kytkettynä.

Koodikatselmointi tärkeää

- Koodikatselmointi ei ole tarkkaan ottaen testausta, vaikka luetaankin joskus staattiseksi testaukseksi.
- Koodikatselmointi on tärkeä lähtökohta aikaiseen laadun varmistukseen.
- Se on sitä tärkeämpää, mitä vaikeammin testattava asia on kyseessä – ja rinnakkaisuus asiat ovat juuri sellaisia.
- Seuraavilla sivuilla on erilaisia rinnakkaisuuteen liittyviä katselmoinnissa tarkistettavia teemoja.
  - Tarkistuslistat ylipäätään ovat hyvä idea katselmoinnin tueksi.
  - Niitä kannattaa rakentaa sovellusaluekohtaisesti ottaen huomioon sen tunnetut teknologiset sudenkuopat.
  - Päivänselvää on se, että kääntäjien varoitustason maksimointi ja staattisten koodin tarkastusohjelmien käyttö ovat suositeltavia.

Koodikatselmoinnin asioita 1/3

- Yleinen koodaustapa:
  - Koodin selkeys… mutkikas koodi on altis rinnakkaisuuden ongelmille.
  - Yleinen robustius-strategia. Asioiden olemassaolo (resurssit, oliot, data), eheys ja funktioiden onnistuminen tarkistetaan koodissa.
  - Arkkitehtuuri. Selkeä tila-arkkitehtuuri. Testattavuus.
- Rinnakkaisuuden toteutus:
  - Kaikkien rinnakkaisuuteen liittyvien asioiden tarkastaminen.
  - Oikea strategia rinnakkaisuuteen – jos sille on vaihtoehtoja. Säikeet vai vuorontaja?
  - Matalan tason rinnakkaisuuden toteuttavien ”patternien” oikea toteutus.
  - Oikea strategia asynkronisiin operaatioihin – jos on vaihtoehtoja. Voiko niitä välttää? (Jos oman koodin kannalta operaatio on ”inline”, käytetään asian synkroniseksi paketoivaa funktiota.)
  - Oikea viestinvälitystekniikka – jos on vaihtoehtoja.

Koodikatselmoinnin asioita 2/3

- Resurssien käsittely:
  - Kriittiset alueet, lukitukset.
  - Turha resurssien lukitseminen.
  - Pitkäaikainen resurssien lukitseminen.
  - Resurssien vapaana olemisen tarkistus ennen omaa allokointia.
  - Defensiivinen strategia resurssien vapaana olemiseen.
  - Resurssien luovutus.
- Suorituskyky:
  - Säikeiden suorituskyky. Mitä kaikkea ja miten niissä tehdään?
  - Raskaat operaatiot – varsinkin synkroniset – voisiko niitä palastella.

Koodikatselmoinnin asioita 3/3

- Toipuminen:
  - Timeoutit.
  - Odotuksista toipuminen. Jos ei onnistukaan…
- Rinnakkaiseen suoritukseen vaarallisten API-funktiokutsujen tunnistaminen.
  - Säieturvallisuus yms.
  - Tähän kannattaa käyttää työkaluja.
  - Sopii integrointitestauksen osaksi – ajetaan tsekkausohjelma koodikannalle.

Katselmointi muillakin järjestelmän abstraktiotasoilla

- Vastaava katselmointi on hyvä tehdä kaikilla systeemin abstraktiotasoilla.
  - Pienimmästä metodista tietokantojen massakäyttöön ja kokonaisarkkitehtuuriin.
- Arkkitehtuurin arviointi katselee kokonaisuutta.
  - Miten hallintaan erilaiset skenaariot:
    - Raskaat kuormitustilanteet.
    - Tietoliikenteen hitaus tai häiriöt.
  - Voi käyttää varsinaisia arkkitehtuurin arvioinnin menetelmiä, mutta jo hyvä tarkistuslistoin tuettu katselmointi on iso etu.

Testauksen luonne

- Tilanteissa, joissa on rinnakkaisuutta, on monia "liikkuvia osia" ja kannattaa ajatella, että kaikkia ilmiöitä ei tunneta ennakolta.
  - Siksi ei voi luottaa vain ennalta suunniteltuihin testitapauksiin.
- Testausta pitää suunnata myös ketterästi havaintojen perusteella. Kun huomataan jotain hassua, tutkitaan, mistä se johtuu – onko se oire isommasta asiasta, jonka vuoksi ohjelma voi kohta räjähtää käsiin.
  - Tämä on ns. tutkivaa testausta, exploratory testing, http://en.wikipedia.org/wiki/Exploratory_testing

Testaustasot

- Testaus on perinteisesti jaettu eri tasoihin, esim.
  - Yksikkötestaus.
  - Integrointitestaus.
  - Järjestelmätestaus.
  - Hyväksymistestaus.
- Rinnakkaisuus näkyy eri tavoilla eri tasoilla.
- Perinteinen yksikkötestaus eristää testattavan asian muista, mutta rinnakkaisuuden testauksessa on tärkeää luoda vuorovaikutuksia!

Testausympäristö

- Tarvitaan tuotantoympäristöä mahdollisimman hyvin vastaava tilanne.
  - Suorituskyky.
  - Järjestelmän kokoonpano ja konfigurointi.
  - Pienikin kokoonpanon muutos voi vaikuttaa.
- Ajoituksen vaihtelu.
  - Erilaisissa ympäristöissä.
  - Erilaisia muita prosesseja.
  - Erilainen systeemin kellotus.
  - Muiden prosessien ja tapahtumien hidastus.
  - Suurempi nopeus – mitä jos prosessori päivitetään?
- Ympäristön parametrien vaihtelu auttaa näkemään muutosten vaikutuksia.

Kuormitus

- Esimerkiksi VR:n systeemien prosessien kerrotaan menneen aivan jumiin, kun tietokannan rivitason lukitukset eskaloituivat taulujen sivutasolle ja taulutasolle (VR:n ongelmat 2011).
- Samaan vaikutti alustan bugi, joka laukesi kovassa kuormitustilanteessa.
- Eli: Testauksessa lisää prosesseja, käyttäjiä, raskautta operaatioihin.
  - Isommat tiedostot, enemmän dataa.
  - Hitaammat verkkoyhteydet.
- Suorituskyky / kuormitustestaus.
  - Testaus erilaisella kuormituksella.
  - Millä prosessien määrällä suorituskyky on vielä ok.
  - Mitä tapahtuu määrää kasvatettaessa.

Robustius

- Terve vainoharhaisuus!
  - Testataan, että oma koodi käsittelee kaikenlaiset ongelmat muissa komponenteissa.
  - Oma säie hallitsee muiden
- Asynkroniset tehtävät.
  - Keskeytymisen hallinta.
  - Poikkeusten käsittelyn testaus.
- Häiriöt muissa komponenteissa.
  - Mockit, simulaattorit testauksen apuna.

Simulointi 1/2

- Simulaattori- ja emulaattoritestaus ei kerro paljoakaan.
  - Nopeus ei vastaa tuotantoympäristöä.
- Mutta!
  - Muiden osajärjestelmien, muiden ohjelmistojen simulointi on arvokasta.
  - Kone- ja automaatiojärjestelmillä simulointi voi liioitella asioita tavoilla, joita ei ole mahdollista kokeilla todellisessa ympäristössä.
  - Tuotetaan tapahtumia satunnaisesti.
  - Tuotetaan häiriöitä – pieni jumi sekoittaa rautatieliikenteenkin.
- Perinteinen testaustyökalu on tynkä, joka toteuttaa toisen komponentin toimintaa simplistisellä tavalla.
  - Jokaisessa tynkätilanteessa kannattaa miettiä monimuotoisemman simulaation rakentamista – ja häiriöiden tuottamista systeemiin sen avulla.
  - Mocki on tässä relevantti käsite: http://en.wikipedia.org/wiki/Mock_object

Simulointi 2/2

- Mallipohjainen testaus.
  - http://en.wikipedia.org/wiki/Model-based_testing
  - Tehdään muista systeemin osista (tila-)malli, jonka suoritusta varioidaan.
  - Tehdään omasta osasesta testausmalli
    - Keskeiset testattavat asiat kattava malli, joka voidaan laittaa vuorovaikutukseen ympäristön kanssa.
- Nopeushaasteita nopeisiin asioihin ”konepellin alla”.

Testauksen aikana havainnoitavaa 1/2

- Tällaisille asioille tarvitaan testitapauksia ja työkaluja, mm. erilaisia monitorointivälineitä.

- ”Näkyvät asiat” – käyttäytyminen.
  - Toteutuvat odotetut asiat ylipäätään:
    - Tapahtuuko kaikki.
    - Lähtevätkö kaikki viestit.
    - Tulostuuko kaikki odotettu näytölle / logiin.
    - Jne…
  - Järjestys – asiat tapahtuvat oikeassa järjestyksessä.
  - Virheettömyys – ei virheitä missään tilanteissa (ajoitus, kuormitus).
  - Aikaan liittyvät havainnot: viiveet, nopeus, nopeuden vaihtelu.
  - Kaikenlaiset ilmiöt kuormitustestauksessa.

Testauksen aikana havainnoitavaa 2/2

- Konepellin alla:
  - Datan eheys siirrettäessä sitä rinnakkaisten prosessien välillä.
  - Poikkeusten käsittely.
- Monitorointi.
  - Kuormitus – prosessori, tietoliikenne, jne…
  - Prosessien, säikeiden yms. määrä.
  - Suoritusnopeus.
  - Puskurien, jonojen vapaa tila ja variaatio.
  - Resurssien käyttö.
  - Suorituksen profilointi.
    - Pullonkaulat.
    - Erot eri tilanteissa.
  - Poikkeukset.

Lopuksi: rinnakkaisuuden testauksen viisi avainasiaa

1. Pienetkin muutokset voivat muuttaa ohjelman käyttäytymistä. Kuun asennosta alkaen…
2. Varsinkin rinnakkaisuusasioissa testaus ei voi todistaa ohjelman oikeaa toimintaa kohdeympäristössä tietyssä tilanteessa, vaan voi vain tuottaa tietoa tilanteista, joissa sen on havaittu toimivan oikein.
3. Testauksen on tässäkin asiassa oltava raakaa ja pyrittävä tuottamaan muutoksia, häiriöitä, satunnaisuutta ja variaatioita ja liioittelemaan asioita, jotta voidaan "varmistaa" ohjelman riittävä robustius.
4. Testaajan on tärkeää olla herkkä erilaisille ilmiöille testauksessa, jotta niiden kautta päästään käsiksi ohjelmassa mahdollisesti oleviin ongelmiin.
5. Koodikatselmointi on tärkeää ja kannattaa tehdä.