# 9. Pouzdanost dolaska poruke

Na ovim vježbama ćemo se susresti s nečim što se na engleskom zove *Message passing semantics* ili *Message delivery guarantee*. Govorit ćemo o tome koliko smo sigurni da je poruka stigla na svoje odredište.

## 9.1 Pouzdanost slanja poruke

U ovih nekoliko tjedana koliko već slušate o raspodijeljenim sustavima, mogli ste primjetiti da poruka ne mora uvijek doći na odredište. Razloga može biti nekoliko:

1. Može se dogoditi da se zahtjev (engl. *request*) izgubi na putu prema odredištu, bio to server ili partnerski čvor. Poruka se može izgubiti iz najbanalnijeg razloga kao što je prekid u vezi s udaljenim računalom. Jedan od načina na koji se ovaj problem može pokušati riješiti je sljedeći - možemo isprogramirati klijent na način da očekuje odgovor (engl. *response*) od servera i ako odgovor ne stigne u određenom vremenskom intervalu ponovno poslati zahtjev. U slučaju da kontinuirano pošiljatelj zahtjeva ne primi odgovor od primatelja, može zaključiti da server ili čvor nisu dostupni.
2. Druga situacija je da se odgovor (engl. *response*) od servera izgubi. Ovaj problem je malo zahtjevnije za riješiti. Mogli bismo pomisliti da je riješenje ponovno osloniti sa na timer. Ovo će raditi ako je operacija koju zovemo na drugom računalu idempotentna - ne uzrokuje nikakve nuspojave (enlg. *side effects*). Razlog tome je sljedeći: zamislimo da se izgubio odgovor, a radimo transfer novca. Klijent ponovno šalje zahtjev za transferom i server ga drugi put izvodi - posljedica toga je da imamo manjak (ili višak) novca na računu. Idempotentne operacije ne uzrokuju nikakve nuspojave, odnosno, nikakve posljedice koje se mogu na bilo koji način opaziti.
3. Još jedan problem koji ćemo spomenuti jest rušenje servera. Ovaj problem rješavamo na način da u trajnu memoriju spremimo nekakav identifikator zahtjeva koje smo izvršili i za svaki izvršeno pošaljemo odgovor.

Kategorije pouzdanosti dolaska, odnosno semantike slanja poruke dijelimo na:

- *at-most-once* - kod ovog pristupa poruke koje se šalju mogu stići primatelju 0 ili 1 put. To znači da je gubitak poruke prihvatljiv. Ovaj pristup je ujedno najbrži i implementacijski najjednostavniji jer se radi o *fire and forget* pristupu.
- *at-least-once* - kod ovog pristupa svaka poruka se šalje 1 ili više puta. Kod ovog pristupa, primatelj može izvršiti zadatak 1 ili više puta. Ovo nije problem kod idempotentnih operacija! Princip rada je jednostavan - imamo timer koji kaže koliko je prošlo od slanja poruke, ako nismo primili odgovor u zadanom vremenskom intervalu, ponovno šaljemo zahtjev.
- *exactly-once* - pristup kojem težimo! Želimo da poruka koju smo jednom poslali točno jednom stigne primatelju. Kod ovog pristupa poruka se ne može izgubiti niti duplicirati. Određenim primjerima se može demonstrirati da ovo nije točno ono što je tokom izrade ovakvog sustava potrebno, već da se želi postići *exactly-once processing*. S druge strane, nećemo ulaziti u to kako se ovo postiže. Osjećamo da je vaćno napomenuti da postoje na internetu resursi kojima se tvrdi da je ovo nemoguće kod određenih sustava.

Iako je *exactly-once* pristup ono što je idealno, nije nam neophodan!

Dovoljno je osigurati da primatelj po primitku poruke točno jednom izvrši zadatak vezan uz poruku. Da bi ovo postigli primatelj mora zapamtiti koje poruke je već obradio kako ih ne bi obradio više puta. Kažemo da želimo *exactly-once processing*.

Probajmo opisati ovaj proces za *exactly-once processing*:

1. zamislimo da imamo primatelja i pošiljatelja;
2. između njih postoji način da poruku prenesemo *at-most-once* semantikom. Dakle, da će poslana poruka najviše jednom doći primatelju;
3. pošiljatelj (klijent u nekim sustavima) je odgovoran za *at-least-once* semantiku. Pošiljatelj mora pamtiti i ponovno slati poruke.
4. *exactly-once* je odgovornost primatelja jer on mora pamtiti što je do nekog trenutku odradio te izbjeći odraditi nekakav posao više od 1 puta.

Naravno, postoje problemi i kod rušenja pošiljatelja, ali u to nećemo ulaziti.

## 9.2 Životni ciklus actora

Putanja u actor sustavu predstavlja mjesto u hijerarhiji na kojoj se actor možda nalazi. Jedna inkarnacija actora je određena putanjom i UID-om, a stvara se pozivom ActorOf metode. U slučaju ponovnog pokretanja actora, instanca objekta se zamijeni na način koji je definiran u Props objektu, ali UID i putanja ostaju nepromijenjeni.

Životni ciklus inkarnacije actora prestaje onda kada se actor zaustavi. Zaustavljanjem actora pozivaju se oodgovarajuće metode vezane uz životni ciklus actora te su svi actori koji prate životni ciklus toga obaviješteni o zaustavljanju. Ako je inkarancija zaustavljena, putanja se može ponovno iskoristiti za kreiranje novog actora. U slučaju ponovnog korištenja putanje (pozivom ActorOf metode) kod novog actora, UID će biti različiti.

ActorRef predstavlja inkarnaciju - putanju i UID. Stoga, ako je actor zaustavljen i kreiran novi s istim imenom, ActorRef stare inkarnacije neće pokazivati na novu.

ActorSelection s druge strane pokazuje na putanju i uopće nije bitno koja je inkarnacija actora. Zbog ovoga se ActorSelection ne može pratiti. Inkarnacija se može dobiti slanjem Identify poruke na ActorSelection, čime se dobiva automatski odgovor Actoridentity.

## 9.3 DeathWatch

DeathWatch služi kako bi jedan actor dobio obavijest kada se neki drugi potpuno zaustavi - NE uslijed greške ili ponovnog pokretanja. Uz pomoć Context.Watch() metode se actor registrira, a uz pomoć Context.Unwatch() deregistrira od praćenja životnog ciklusa nekog drugog actora.

Npr. actor a želi pratiti životni ciklus actora b - actor a mora pozvati Context.Watch(b).

U slučaju da se b zaustavi u potpunosti, a dobiva *Terminated* poruku. Ta poruka je ugrađena u actor sustav. Ovu poruku će dobiti čak i actori koji se registriraju nakon što actor b prekine s radom.

Metode životnog ciklusa:

1. PreStart - metoda koja se poziva odmah nakon pokretanja actora. Prilikom restarta, ova metoda se zove iz metode PostRestart.
2. PreRestart - recimo da se actor ponovno pokreće usljed pogreške koja se dogodila; preRestart metoda se izvršava nad starom instancom actora i sadrži informaciju o exceptionu i poruci koja ga je uzrokovala. Naravno, ukoliko se pogreška nije dogodila usljed primanja poruke, onda poruka neće biti dostupna. Ova metoda je najbolje mjesto za čišćenje resursa koje actor koristi. Po defaultu zaustavlja svu djecu i poziva PostStop metodu. Nakon toga inicijalna ActorOf metoda se koristi za izradu nove instance.
3. PostRestart - nastavak na priču o PreRestart - Zatim se novom actoru pozove PostRestart metoda te sadrži exception koji je uzrokovao restart. Po defaultu, iz PostRestart metode se zove PreStart kao kod normalnog pokretanja. Važno je napomenuti da Restart ne utječe na sadržaj poštanskog pretinca (engl. *mailbox*). Poruka koja je prouzročila restart se neće ponovno primiti.
4. PostStop - metoda koj se poziva nakon zaustavljanja actora. Može se koristiti za odregistriranje od različitih servisa.

## 9.4 Izvori

1. `http://doc.akka.io/docs/akka/snapshot/general/message-delivery-reliability.html`, 04.05.2016. u 00:09.
2. `https://brooker.co.za/blog/2014/11/15/exactly-once.html`, 04.05.2016. u 00:28.
3. `http://getakka.net/docs/Actor%20lifecycle`, 04.05.2016. u 01:07
4. `http://doc.akka.io/docs/akka/current/scala/actors.html`, 04.05.2016. u 01:07.