5. Akka - Detaljnije

Na prethodnim vježbama smo se upoznali s osnovama Akka.NET frameworka. Kreirali smo prvi actor sustav, prvoga actora i razmijenili poruke među istima. Na ovim vježbama nastavljamo tamo gdje smo stali s akka-om. Upoznat ćemo se sa detaljima kako actor funkcionira interno, hijerarhiju actor sustava, putanjama i HOCON konfiguracijskim postavkama.

5.1 Hijerarhija

Na prošlim vježbama smo kreirali ActorSystem ali nismo ništa detaljno o njemu komentirali. Važan koncept prilikom rada s actor-ima i općenito s distribuiranim sustavima jest kako pronaći dio sustava koji želimo koristiti. Na sreću, akka ovo ima dobro riješeno. Već smo rekli da je ActorSystem hijerarhijska organizacija actora, a sada ćemo reći par riječi o toj organizaciji.

Važno je znati da svaki actor ima roditelja, a neki actori uz to imaju djecu. Roditelji nadziru svoju djecu. Budući da roditelji nadziru svoju djecu, to znači da **svaki actor ima nekog tko ga nadzira i svaki actor može postati netko tko nadzire**. Unutar hijerarhije actor sustava postoje actori koji su direktno u nadležnosti samo actor sustava, a postoje i djeca actori koji su u nadležnosti drugih actora.

Hijerarhijska struktura actor sustava je dana na slici 5.1.

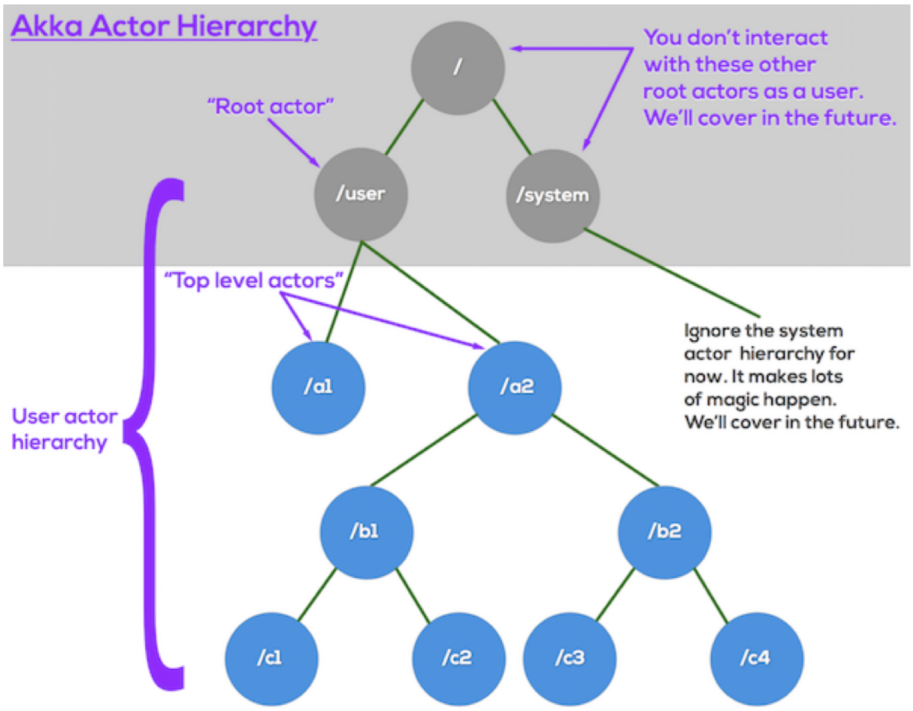

Figure 5.1: Hijerarhija actor sustava detaljno

Na samom vrhu hijerarhije postoje 3 actora koja se zovu "Guardians". Ta tri actora su korijeni cijelog actor sustava.

Actor s putanjom "/" je bazni actor cijelog actor sustava i možemo ga nazvati "Root Guardian Acto". Ovaj actor nadgleda "/user" i "/system" actore. Svi actori osim ovoga imaju nekoa tko ih nadgleda.

Actora s putanjom "/system" nazivamo "The System Guardian". Osnovni posao ovog actora je osigurati da se sustav pravilno izgasi te da održava, odnosno nadzire, druge actore koji implementiraju i pružaju svojstva i pomoćne funkcionalnosti na razini frameworka.

Actora s putanjom "/user" nazivamo "The Guardian Actor" i pod njim se stvaraju svi actori koje kreira korisnik. Slika 5.2. Općenito se može naći da se ovaj actor naziva "root actor".

Pod "/user" actorom se stvaraju svi actori koje programer definira. Djeca ovog actora se zovu "top level actors". Ponovno, actori se uvijek kreiraju kao djeca nekog actora. Svaki put kada kreiramo actora direktno iz actor sustava, stvoreni actori će postati djeca ovog actora, primjer 5.1.

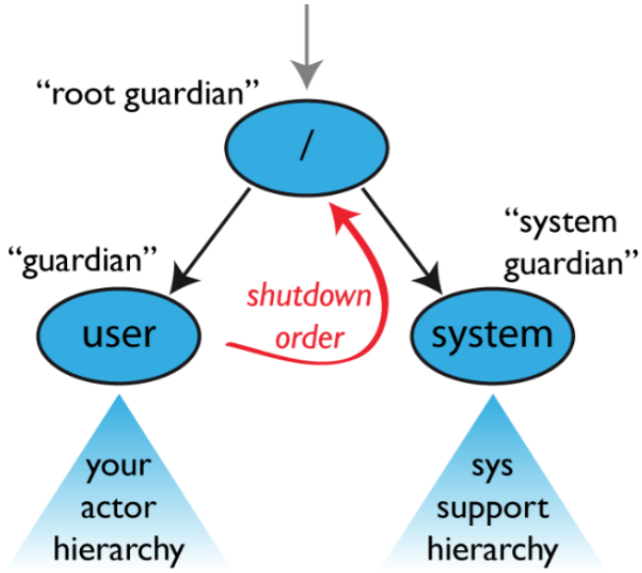

Figure 5.2: Hijerarhija actor sustava ukratko

Primjer 5.1: Kreiraj top level actore

```
IActorRef a1 = MyActorSystem.ActorOf(Props.Create<BasicActor>(),
    "a1");
IActorRef a2 = MyActorSystem.ActorOf(Props.Create<BasicActor>(),
    "a2");
```

Ukoliko želimo kao što je na slici napravljeno kreirati actore koji će biti djeca actora "a2", onda to napravimo tako da unutar actora uz pomoć njegovog Context-a kreiramo nove actore, na primjer 5.2.

Primjer 5.2: Kreiraj djecu actora a2

```
// unutar actora a2 kreiraj djecu:
IActorRef b1 = Context.ActorOf(Props.Create<BasicActor>(), "b1");
IActorRef b2 = Context.ActorOf(Props.Create<BasicActor>(), "b2");
```

5.2 ActorPath

U prethodnom poglavlju smo spomenuli da su "/", "/system", "/user" putanje guardian actora, odnosno, triju actora koji su na najvišoj razini actor sustava.

Svaki actor ima svoju adresu i svakom actoru se može poslati poruka preko njegove adrese. Putanja je samo dio te adrese. Primjer potpune adrese:

Primjer 5.3: Primjer pune adrese actora

```
akka.tcp://MySystem@localhost:9001/user/a2/b1
```

Pogledajmo detaljno komponente adrese:

- akka.tcp - ova porcija adrese određuje koji protokol se koristi u komunikaciji;

- MySystem - ime actor sustava koje je pridjeljeno prilikom kreiranja actor sustava;
- localhost:9001 - adresa računala na mreži skupa s portom na kojem sluša;
- /user/a2/b1 - putanja koja precizno određuje gdje u hijerarhiji se nalazi traženi actor.

5.3 HOCON

HOCON (Human-Optimized Config Object Notation) je fleksibilan i proširiv konfiguracijski format. Akka.NET koristi HOCON za konfiguriranje actora i actor sustava.

HOCON-om se nećemo detaljno baviti, spominjat ćemo ga samo koliko nam je potrebno za određene funkcionalnosti koje su nam potrebne.

Sljedeća dva primjera prikazuju dovoljnu konfiguraciju za izradu ditribuiranu sustav koristeći Akka.Remoting.

Primjer 5.4: HOCON config za udaljeni actor sustav

```
akka {
        # ovaj dio konfiguracije se referencira kao akka.actor
        actor {

                # uocite dvaput dvostruke navodnike - ovo je samo ako definiramo
                # konfiguraciju unutar string-a u C\# kodu.
                # ako stavljamo konfiguraciju u App.config, dovoljno su jednom
                # dvostruki navodnici
                # RemoteActorRefProvider - je potreban da osigura da actori
                # koji su referencirani na razlicitim sustavima ukljuce svoje
                # informacije za udaljeni pristup
                # iz dokumentacije - RemoteActorRefProvider - Starts up actor on remote
                # node and creates a RemoteActorRef representing it.
                provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
        }

        # ovim dijelom konfiguriramo Akka.Remote modul
        remote {

                # ugradeni modul za tcp komunikaciju
                helios.tcp {

                        # port na kojem slusamo
                        port = 12000
                        # adresa
                        hostname = localhost
                }
        }
}
```

Primjer 5.5: HOCON config za lokalni actor sustav

```
akka {

        actor {

                provider = ""Akka.Remote.RemoteActorRefProvider,
                    Akka.Remote""
        }

        # ovim dijelom konfiguriramo Akka.Remote modul
        remote {

                # ugradeni modul za tcp komunikaciju
                helios.tcp {

                        # port na kojem slusamo
                        port = 12001
                        # adresa
                        hostname = localhost
                }
        }
}
```

5.4 Actor - Detaljno

U ovom poglavlju spominjemo detaljnije malo od čega se actor sastoji. Actor sam po sebi je spremni za stanje (*engl. State*), pretinac za poštu (*Mailbox*), djecu (*Children*) i strategiju nadzora (*engl. Supervisor Strategy*).

5.4.1 ActorReference

Kao što smo već naveli, prilikom stavaranja actora ne dobivate referencu na actora direktno, već referencu ActorRef koji zna poslati poruku točno onom actoru kojeg želimo. Da bismo uživali sve prednosti actor modela, actor mora biti zaštićen od utjecaja vanjskog svijeta. Stoga su actori prikazani prema vanjskome svijetu kao reference, odnosno *ActorRef*. Ovaj ActorRef je objekt kojim možemo slobodno baratati (kao slanje u konstruktor na prethodnim vježbama). Ova podjela na unutarnji i vanjski objekt nam omogućuje transparentnost u sljedecim segmentima:

- možemo restartati actor bez potrebe da osvježim reference na drugim mjestima - imamo referencu na ActorRef koji će ostati isti objekt;
- stavljanje actora na udaljenog poslužitelja;
- slanje poruka actorima koji se nalaze na udaljenim aplikacijama.

Najvažnija prednost ovoga je ta što ne možemo pristupiti stanju actor-a izvana, osim ako programer ne napravi kardinalnu grešku prilikom programiranja actora.

5.4.2 State

Da bi nam actori bili korisni, moraju sadržavati nekakvo unutarnje stanje. To mogu biti varijable, zahtjevi koji su na čekanju, itd. Ovi podatci su ono zbog čega je actor vrijedan i potrebno ih je zaštiti od vanjskog svijeta. Kako ne bi iz nekog drugog dijela koda zabunom, npr. izbrisali neki podataka ili ga izmijenili. konceptualno svaki actor ima svoju, s aspekta zahtjeva resursa, laganu nit. Kažemo konceptualno, jer interno ovo nije istina, ali o možemo razmišljati o njima kao da su svaki na svojoj niti. Te niti su zaštićene od ostatka sustava. To znači da programer ne mora programirati s lockovima i brinuti oko sinkronizacije pristupa.

U pozadini Akka.NET će pokreunuti skupine actora na više niti, ali gdje više actora može dijeliti istu nit, a uzastopni pozivi istom actoru se mogu izvršiti na različitim nitima. Akka.NET brine da ovo funkcionira i da ne utječe na to da samo jedna nit u svakom momentu ima pristup stanju actora.

S obzirom da je unutarnje stanje izrazito važno za actora i njegove operacije. Nekonzistentna stanja su fatalna i dovode do ponovnog pokretanja actora. Pri tome će se stanje actora izgraditi nanovo otpočetka, kao da je actor prvi put pokrenut. Ovo je da se sustavu omogući da samog sebe izliječi.

Postoje načini da se stanje actora vrati na stanje u kojem je bio prije rušenja, ali o tome nekom drugom prilikom.

5.4.3 Behavior

Ponašanje je mapiranje primljene poruke s funkcijom. Na prošlim vježbama smo ga definirali unutar konstruktora actora tako što smo u Receive metodu slali lambda izraze kojima smo definirali što učiniti s pojedinim tipom primljene poruke. Dakle, actor primljenu poruku izjednači s definiranim ponašanjima i ako ima ponašanje definirano za taj tip poruke, actor pozove pridruženu funkciju. Ovo ponašanje se može mijenjati tokom vremena, ali o tome nekom drugom prilikom.

5.4.4 Mailbox

Svrha actora je procesiranje poruka, a ove poruke su poslali drugi actori, ili su primljene od izvan actor sustava. Svaki actor ima točno jedan mailbox, tj. pretinac pošte, i svi pošiljatelji šalju svoje poruke u taj pretinac. Unutar mailboxa poruke se stavljaju u nekakvu kolekciju. Poruke koje se nalaze u mailboxu se slijedno procesiraju. Za poruke koje šalju različiti actori ne možemo garantirati redosljed, ali za poruke koje šalje jedan actor možemo garantirati da će stići istim redosljedom kojim su poslane.

Postoje različite implemnetacije mailboxa i programer može odabrati koji mailbox će actor koristiti prilikom stvaranja istog. Defaultno zadan mailbox je FIFO struktura, dakle, prva poruka koja se primi je prva poruka koja će se obraditi. Naravno, nekad ne želimo da actor procesira poruke po redosljedu primitka poruke, već po nekom zadanom prioritetu. U tom slučaju možemo koristiti prioritetni mailbox koji nove poruke ne stavlja na kraj strukture već po prioritetu na odgovarajuću poziciju.

Važna postavka Akka.NET-a u odnosu na ostale implementacije actor modela je ta što trenutno ponašanje mora uvijek pokušati procesirati sljedeću poruku u mailboxu stoga nema skeniranja mailboxa za nekom porukom koja odgovara. Nemogućnost da trenutno ponašanje procesira poruku će se tretirati kao neuspjeh, osim ako ponašanje nije overrideano.

5.4.5 Supervisor Strategy

Posljednji dio actora je način na koji rješava problem pogrešaka kod djece. Obnova sustava se zatim, na osnovu strategije, automatsku izvrši. Ova strategija se ne može mijenjati nakon što je actor kreiran.

5.4.6 Kada se actor ugasi?

Jednom kada se actor izgasi "prirodnim" putem, dakle, ne usljed pogreške već pozivom metode Stop, actor oslobodi zauzete resurse, a sve poruke koje su ostale u mailboxu se šalju sustavu u pretinac "Dead letter mailbox". Nakon toga se sve poruke koje se pokušaju poslati tom actoru preusmjeravaju u taj sistemski mailbox.

5.5 Dispatcher - ukratko

Kako bismo mogli rezultat actora zapisati na windows formu, nužno je da je actor pokrenut na istoj niti. Ovo možemo postići postavljanjem drugog dispatchera.

Dispatcher je zadužen za pokretanje bilo kakvog koda koji se pokreće u actor sustavu. Oni su jedan od najvažnijih dijelova Akka.NET-a jer kontroliraju propusnost i vrijeme koje svaki actor dobiva. Defaultno zadani dispatcher je Global Dispatcher. Najčešće je on dovoljan za potrebne zadatke. Global dispatcher se vrti nad ThreadPoolom.

Postoje situacije u kojima nam je potreban drugi dispatcher ako želimo, npr. kontrolirati koliko vremena će scheduler pokrenuti kojeg actora. O scheduleru nećemo sada, ali generalnu ideju možete dobiti.

Drugi slučaj korištenja je ako želimo namjestiti sustav na način da actori koji se inače dugo izvršavaju ne uzrokuju "izgladnjivanje" ostatka sustava.

Dalje, možemo osgiurati da bitni actori uvijek imaju svoju nit.

Primjer koji vam je dostupan na moodleu koristi SynchronizedDispatcher da bi stvorio actora i pokrenuo ga na istoj niti koja ga je stvorila. U našem slučaju, UI nit. To nam je bitno da bismo mogli osvježiti UI elemente.

Kojeg dispatchera će actor koristiti konfiguriramo uz pomoć HOCON-a.

Doduše, za postavljanje jednome actoru samo dispatcher, ne moramo koristit HOCON već props objekt posjeduje metodu WithDispatcher() kojoj pošaljemo u obliku stringa koji dispatcher da koristi, npr. "akka.actor.syncrhonized-dispatcher". Primjer 5.6.

Primjer 5.6: Postavi dispatcher

```
1 Props props = Props.Create(...)
2 .WithDispatcher("akka.actor.synchronized-dispatcher")
```

5.6 Zadatci

1. Na moodle sustavu imate primjer windows forms aplikacije koja šalje zahtjeve wikipediji. Jedna od komponenti na osnovu unesenog teksta u polje šalje zahtjev pretraživanju wikipedije, a drugi na osnovu odabranog rezultata pretrage dohvaća stranicu. Izmijenite ovaj primjer na način da 1 actor pošalje zahtjev searchu wikipedije na osnovu korisnikova unosa. Natrag biste trebali dobiti listu rezultata koju ćete prikazati na formi (zovimo ju ListResult). Svaki od dobivenih rezultata pošaljite novom actoru koji mora poslati zahtjev na wikipediju za stranicom. Svi svoje rezultate vraćaju nekome. Na formi se prikazuje samo ona stranica koja je odabrana u listi. Razmislite koliko vam actora treba te kako ih organizirati. Proučite kako radi metoda Become, možda će vam koristiti. Ovdje nemate server dio.

6. Akka Remoting

U prethodnim poglavljima smo odradili uvod u Akka.NET i obradili putanje, HOCON, hijerarhiju sustava. Ovdje će biti riječ o Akka.Remote.

6.1 Akka.Remote

Akka.NET projekt, osim osnovnog, ima dosta dodatnih paketa kojima se prosiruju mogucnosti Akka-e. Jedan od tih je Akka.Remote, a posebno ga izdvajamo i obradjujemo jer nam omogucuje izradu actor sustava koji se nalazi na razlicitim racunalima na mrezi. Dakle, ovaj dodatak je jedan od onih koji nam omogucuje izradu uistinu distribuiranog sustava.

Mogucnosti Akka.Remote ukljucuju:

1. *Location transparency with RemoteActorRef* - sto se programera tice, ne postoji razlika izmedu programiranja komunikacije izmedu dva lokalna actora te komunikacije lokalnog i udaljenog actora. To znaci da je lokacija actora potpuno transparentna s obzirom na kod. Potrebno je samo podesiti neke konfiguracijske postavke.
2. *Remote addressing* - Akka.Remote prosiri mogucnosti Address i ActorPath komponenti Akka.NET-a da ukljuce informacije kako se povezati s udaljenim procesima koristeci ActorSelection.
3. *Remote messaging* - Transparentno slanje poruka actorima koji pripadaju drugim actor sustavima na mrezi.
4. *Remote deployment* - slanje actora na udaljeni actor sustav bilo gdje na mrezi. Lokacija actora na mrezi postaje implementacijski detalj u Akka.Remote.
5. *Multiple network transports* - Akka.Remote dolazi s podrskom za TCP, ali moze podrzati razlicite transporte i aktivira vise razlicitih istovremeno.

Sve u Akka.NET je dizajnirano s mislju na distribuirani sustav. Sve funkcionalnosti su jednako dostupne kada je sustav pokrenut na jednom racunalu ili klasteru od stotinu racunala.

Vazno je imati na umu da sva interakcija koja se dogadja preko mreze je potpuno asinkrona. To znaci da moze potrajati i nekoliko minuta da poruka dodje do primatelja. Takodjer, vjerojatnost da ce se poruka izgubiti kada se salje preko mreze je mnogo veca.

Postavlja se pitanje - sto je s velicinom poruke? Koja je najveca poruka koju mogu poslati?

Zadana maksimalna velicina poruke je 128 KB (najmanje 32 KB). Naravno, uz pomoc HOCON konfiguracije moze povecati maksimalna velicina poruke, ali do najvise 4MB. Poruke koje su vece od 4 MB se odbacuju, a za slanje neke datoteke koja je veca, potrebno je pronaci i isprogramirati nacin na koji ce se te vece datoteke poslati dio po dio (u fragmentima).

Primjer 6.1: Izmjena maksimalne velicine poruke

```
akka {
    helios.tcp {
        # Maximum frame size: 4 MB
        maximum-frame-size = 4000000b
    }
}
```

6.2 Peer-to-Peer vs Client-Server

Akka.Remoting je komunikacijski modul koji sluzi za povezivanje actor sustava u Peer-to-peer konfiguraciju. U slucaju kada koristimo Akka.Remoting u klijent server konfiguraciji, razliku izmedju jednog i drugog radimo sami kroz implementaciju, a ne Akka.NET. Za konfiguracije klijent-server preporuka se koristenje Akka I/O modula.

Izrada ovog modula je vodena dvjema odlukama:

1. Komunikacija izmedu ukljucenih sustava je simetricna: ako se sustav A poveze sa sustavom B, onda se se sustav B mora moci povezati sa sustavom A.

2. Uloge komunikacijskih sustava su simetricne s obzirom na model konekcije: nema sustava koji koji sam prihvaca konekcije i nema sustava koji samo inicira konekcije. Zbog ovoga nije moguce na siguran nacin stvoriti ciste klijent-server konfiguracije s predefiniranim ulogama.

Akka.Remoting vecinski sluzi kao osnova za Akka.Cluster o kojem ce biti rijec u sljedecem poglavlju. Koristenje samog Akka.Remoting modula je ograniceno na slucajeve koji ne zahtjevaju elasticnost i otpornost na pogreske koje Akka.Cluster nudi.

6.3 Kako ukljuciti Akka.Remoting?

Primjer Hello world aplikacije koja koristi Akka.Remoting je dostupan na sustavu moodle.

Ovdje cemo navesti korake ukratko:

1. Instlirati Akka.Remote nuget packet
2. Konfigurirati RemoteActorRefProvider putem HOCON konfiguracije
3. Omoguciti barem 1 port za konekciju
4. Konfigurirati adresu za svaki transport
5. Pokrenuti actor sustav

Primjer 6.2: HOCON Server

```
akka {
    actor {
        provider = "Akka.Remote.RemoteActorRefProvider,
            Akka.Remote"
    }
    remote {
        helios.tcp {
            port = 8081 #bound to a specific port
            hostname = localhost
        }
    }
}
```

Primjer 6.3: HOCON Klijent

```
akka {
    actor {
        provider = "Akka.Remote.RemoteActorRefProvider,
            Akka.Remote"
    }
    remote {
        helios.tcp {
            port = 0 # bound to a dynamic port assigned by the OS
            hostname = localhost
        }
    }
}
```

HOCON konfiguracije za klijenta i servera su dane u 6.3 i **??**.

Povise smo spomenuli nekoliko pojmova koje cemo ovdje poblize objasniti:

- Transport - transport se odnosi na mrezni protokol za prijenos podataka, npr. TCP ili UDP. Zadani transport za Akka.NET je Helios TCP transport. Transport je potrebno povezati s IP adresom i portom, sto smo napravili u obje HOCON konfiguracije povise unutar helios.tcp sekcije.

- Adresa - odnosi se na kombinaciju IP + port.
- Endpoint - predstavlja krajnju tocku u komnikaciji koja podrazumjeva koristenje nekog protokola i povezanost s odredenom adresom. Npr. ako otvorimo TCP transport na adresi `localhost:8888`, onda smo kreirali endpoint za TCP transport na toj adresi.
- Asocijacija - veza izmedu dva endpointa, gdje svaki endpoint pripada razlicitom actor sustavu. Da bi se asocijacija mogla kreirati potrebno je imati validni ulazni endpoint i validni izlazni endpoint.

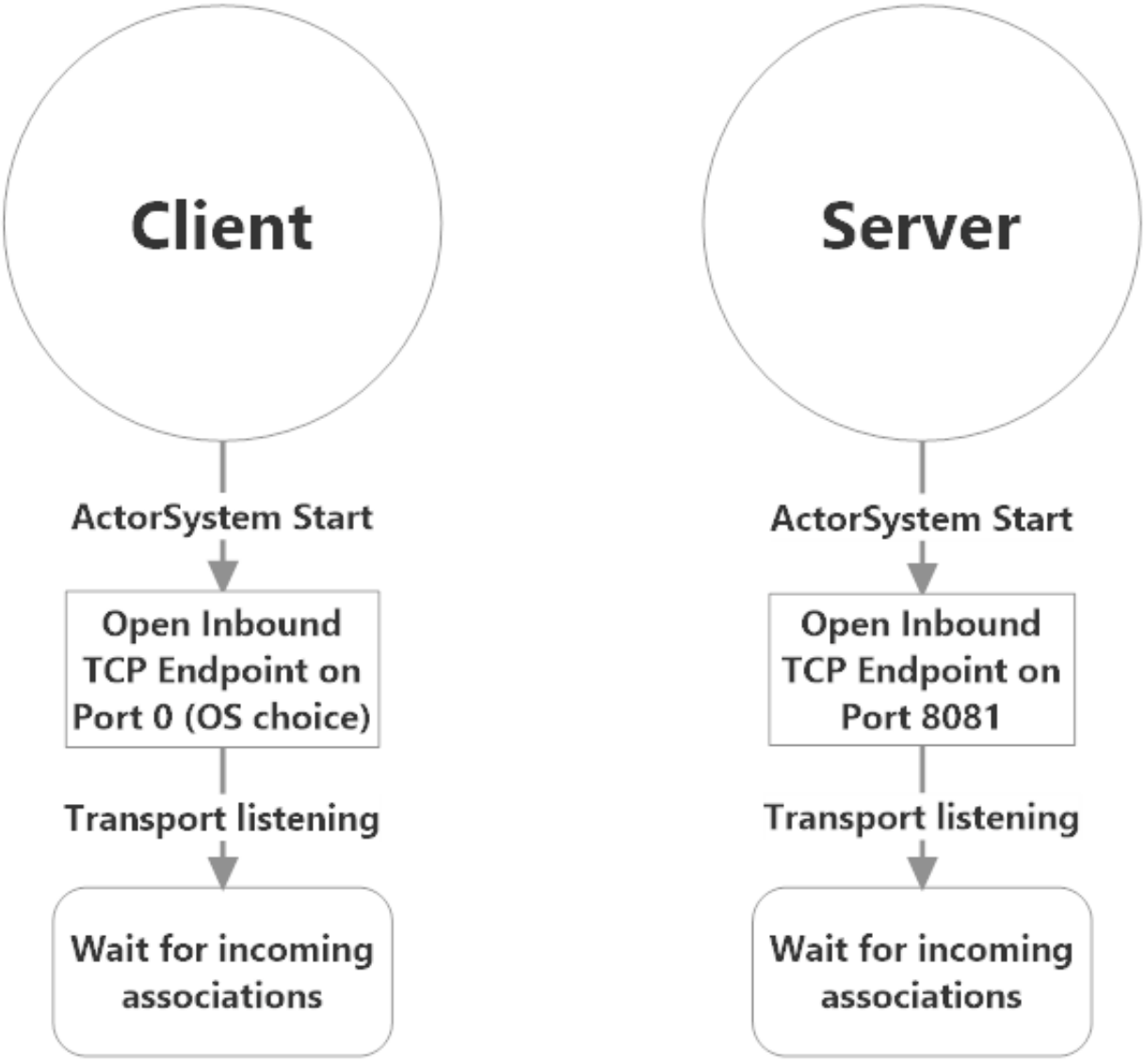

Figure 6.1: Dva pokrenuta sustava

Slika 6.1 prikazuje stanje dvaju actor sustava u trenutku pokretanja. Oba actor sustava se pokrenu, otvore svoj transport i povezu ga s konfiguriranom adresom. Povezivanjem transporta s konfiguriranom adresom stvara ulazni endpoint za svaki sustav. Nakon toga, sustavi jednostavno cekaju da jedan od njih inicira konekciju. Asocijacija izmedju dva sustava se formira kada bilo koji cvor pokusa stupiti u kontakt s drugim.

Proces stvaranja asocijacije zapocinje kada jedan od cvorova, npr. klijent, posjeduje actora koji zeli poslati poruku ActorSelection-u koji se nalazi na Serveru.

RemoteActorRefProvider koji je ugraden u prvi sustav ce provjeriti postoji li otvorena konekcija prema drugom sustavu u trenutku kada primjeti udaljenu adresu u ActorSelection-u. Ako ne postoji, onda ce otvoriti novi izlazni endpoint koristeci TCP transport i poslati "handshake" poruku drugom sustavu. Interno, za TCP transport se koristi TCP soket.

Drugi sustav prima handshake poruku, prihvaca ulaznu asocijaciju i odgovara sa "associate" porukom koja signalizira da je proces stvaranja asocijacije gotov.

Prvi sustav nakon toga salje poruku actoru na drugom sustavu.

Ovo je ukratko kako asocijacije funkcioniraju.

6.4 Zadatci

1. U prethodnim vježbama ste imali napraviti 2 actora koja će se dopisivati porukama Ping i Pong. Napravite klijentskog actora koji će serveru slati poruku Ping, a server će odgovarati porukom Pong.
2. Izmijenite prethodni zadatak na način da imate 2 klijenta koji se dopisuju porukama Ping i Pong preko istog servera. Neka svaki od actora bilježi svaki put kada primi poruku (tj. broji primljene poruke). NB: Ako u HOCON-u za port postavite 0, Akka.NET će sama dodijeliti slobodni port klijentu.
3. Prethodni zadatak izmijenite na način da koristite windows forms aplikaciju za klijente umjesto server aplikacije.
4. Napravite jednostavnu chat aplikaciju u kojoj klijenti komuniciraju preko istog servera. Svaki klijent mora imati jedinstveno ime koje unosi korisnik. Korisnik ne može poslati poruku ostalim korisnicima dok ne unese jedinstveno ime, drugim riječima, ne može pristupiti chat sobi. Razmislite o nekoliko stvari:
   - Kako organizirati ovu aplikaciju na serveru i klijentu, koliko različitih actora je potrebno?
   - Kako su actori organizirani?
   - Koje poruke actor mora podržati?
   - Koliko poruka je potrebno razmijeniti prije nego se klijentu dozvoli pristup u chat sobu?

6.5 Izvori

1. http://getakka.net/docs/remoting/, 26.04.2016., 14:58