UPPSALA
UNIVERSITET

UPTEC STS 19047
Examensarbete 30 hp
November 2019

Actor model evaluation

The impacts and possible enhancement of performance and scalability of a platform

Sofie Mähler

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Actor model evaluation

Sofie Mähler

The aim of this project is to study how a change to an actor-model can affect an existing software system. The main emphasis is on quantifying the advantages in terms of performance and understandability of the code that such a change can bring. In this project, actors have been used in two different actor-models, the Akka.NET actor-model and Service Fabric Reliable Actor. An actor is a unit with a specific task, a unit that only perform its only task. The actor saves its current state and communicates with other actors through their mailboxes.
The two actor models were studied, and prototypes made before the model with Service Fabric Reliable Actor was implemented into the existing system. The choice fell on Azure's Service Fabric Reliable Actor mostly because the current system was based on a service from Azure which made it a more appropriate model to implement. Comparisons of the code were made before and after the implementation of actors into the software system, with the purpose of study the possible change of understandability of the code. The performance in faster results was measured by a reiteration of selected function calls before and after the implementation of the actors.¨
In conclusion, the actor-model had to be adjusted to fit the structure of the existing software system it was implemented into. The code became easier to get an overview of and therefore the understandability increased, the project also concluded the performance could be enhanced when the function calls was repeated enough times.

Handledare: Henrik Doverhill
Ämnesgranskare: Konstantinos Sagonas
Examinator: Elísabet Andrésdóttir
ISSN: 1650-8319, UPTEC STS19 047
Tryckt av: Uppsala

Sammanfattning

Syftet med projektet är att undersöka hur ett skifte till en actor-modell skulle kunna påverka ett system. Detta med fokus på systemets prestanda i form av möjlighet till snabbare resultat och enklare förståelse för koden i form av en enklare modell. Det syftet planerades att uppnås genom att undersöka två olika actor-modeller för att sedan göra en enklare implementation av den typ av actorer som ansågs passa bättre ihop med Caspecos lönemotor. En implementation av ett antal Service Fabric Reliable Actors gjordes i delar av lönesystemet. Valet föll på Microsoft Azures Service Fabric Reliable Actors till stor del eftersom lönemotorn redan var baserad på en service från Microsoft Azure och detta ansågs vara det mer lämpade alternativet att implementera på ett smidigt vis.

Företaget Caspecos lönemotor är idag ett molnbaserat system där det är av största vikt att alltid kunna visa exakta och aktuella siffror. Det finns i företaget ett intresse av att undersöka hur en actor-modell skulle kunna påverka prestandan i form av enklare cache-minne och snabbare resultat. Ytterligare ett intressant område som en actor-modell skulle kunna påverka vore om modellen skulle kunna förenkla den mentala bilden av systemet eftersom lönemotorn idag kan upplevas som komplex.

En actor är en enhet som har i en strikt tolkning i uppgift att hantera en sak och en sak endast. Sitt nuvarande tillstånd lagrar en actor i sitt så kallade state. Den kommunicerar med andra actorer i ett actor-system genom meddelanden till varandras brevlådor. Det finns ett antal olika actor-modeller och programmeringsspråk där actorer kan användas.

Hur koden förändrats undersöktes genom att en del av koden jämfördes innan och efter implementerandet av en actor i den delen av systemet. Implementeringen resulterade i fler filer och aningen längre kod, dock var varje fil mer enkelspårig i vad den gjorde och kunde på det viset tolkas som enklare att förstå sig på.

Förändring i prestanda mättes genom att fyra olika typer av anrop (innan och efter införandet av en Service Fabric Reliable Actor) gjordes upprepade gånger och mättes i avseende på anropslängd. Varje typ av anrop mättes i fyra tester med 10 000 anrop per test, med hjälp av dessa data beräknades väntevärden och varians för testerna. Den statistiken låg till grund för jämförelserna som visade på att första anropen i varje test tog flera gånger så lång tid än de följande både i testerna med och utan actor. För de flesta tester med actorer tog första anropen mycket längre tid även fast de följande anropen var snabbare. De tog så pass lång tid längre att det ansågs krävas orealistiskt många anrop för att tiden för det första anropet skulle tjänas in. Anropen med hjälp av en Service Fabric Reliable Actor hade i nästan alla fall en lägre varians som gav mer förutsägbara längder på anropen. En slutsats blev därför att Service Fabric Reliable Actors skulle kunna gynna systemet om tiden för första anropet kunde kortas.

Innehållsförteckning

1. Inledning

Företaget Caspeco är ett Uppsala-baserat företag som tillhandahåller digitala verktyg och mjukvarusystem för bland annat lönehantering, schemaläggning och bokning. Caspecos lönesystem är idag molnbaserad genom Microsoft Azure och bygger på att alltid kunna visa aktuella och exakta siffror. Företaget är intresserade av att undersöka möjligheterna med att använda sig av en actor-modell, främst för att undersöka möjligheterna att öka prestandan och på sikt även skalbarheten. Några områden som är extra intressanta är hur en actor-modell skulle kunna påverka cache-minnet, den mentala modellen av systemet och en ökad prestanda i form av snabbare resultat. Anledningen till att en förenklad modell över det nuvarande systemet vore intressant är att koden för lönemotorn idag kan uppfattas som komplex och bitvis snårig.

Actor-modellen är ett sätt som ett program kan skrivas på för att försöka förbättra olika aspekter av programmet. Ett problem företaget Caspecos lönesystem har var att det var svårt att förstå sig på strukturen och arbeta med koden. En tanke fanns att det kunde underlättas med hjälp av en actor-modell. Genom att göra en del tester med en actor-modell på den aktuella koden skulle det kunna gå att undersöka om det skulle kunna underlätta förståelsen av koden. Actor-modellen skulle dock påverka mer än bara förståelsen av koden, den skulle förmodligen också påverka på andra vis och bland annat förändra systemets prestanda.

En actor definieras enligt Agha (1986) som en liten enhet som har till uppgift att göra en specifik uppgift. Varje actor har en mailadress och ett beteende (Agha, 1986). Enligt Garnock-Jones (2016) togs actor-modellen fram i början av 1970-talet av främst Carl Hewitt och har sedan dess vidareutvecklats av ett flertal aktörer. Agha's arbete tog den tidiga actor-modellen vidare till samtidig objektorienterad programmering, Agha bidrog även till att actor-modellen började behandlas som en generell programmeringsmetod. Idag finns en uppsjö av olika implementationer av olika versioner av actor-modellen (Garnock-Jones, 2016). I detta projekt har två olika actor-modeller använts, nämligen Akka.NET och Service Fabric Reliable Actor.

1.1 Syfte och mål

Syftet med projektet är att undersöka hur ett skifte till en actor-modell skulle kunna påverka ett system. Detta med fokus på systemets prestanda i form av möjlighet till snabbare resultat och enklare förståelse för koden i form av en enklare modell.

1.2 Avgränsningar

Med hänsyn till projektets omfattning har en förenkling av lönemotorn gjorts där ett antal utvalda delar av lönemotorn undersökts. Komplexitet i lönemotorn medförde även att undersökningen av actor-modellens påverkan genomfördes av utvalda delar av lönemotorn och ej på hela systemet. På grund av svårigheter med åtkomst till källkod

för lönemotorn skedde undersökningar av skillnader mellan två actor-modeller genom simulerade lönekörningar i lösningar utanför den faktiska lönemotorn.

2. Bakgrund

2.1 Definition av en actor och actor-modellen

Garnock-Jones (2016) menar på att det finns två olika typer av objekt inom actor-modellen, actorerna själva och meddelanden. Meddelandena är oföränderliga data som skickas mellan actorerna. Actorerna beskriver han som enheter med ett state som endast kan påverka varandra genom meddelandena. Garnock-Jones beskriver en actors state som analogt med instansvariabler som inte går att påverka utifrån. Utöver state skriver Garnock-Jones även att alla actorer har ett interface som anger och hanterar vilka meddelanden actorn kan svara på (Garnock-Jones, 2016). Enligt Cordemans, Steegmans och Boydens (2015) brukar en actor syfta till en parallell enhet som reagerar på meddelanden. Reaktionen på ett meddelande kan leda till att actorn; skickar ett eller flera meddelanden, skapar en eller flera ny actorer eller ändrar sitt eget beteende inför framtida meddelanden. Nya meddelanden som anländer till actorn hamnar i en brevlåda i form av en kö som actorn tar ett meddelande i taget ifrån för att hantera (Cordemans, P., Steegmans, E. & Boydens, J., 2015). Microsoft Dokumentation (2017b) definierar en actor som en enkeltrådad, oberoende och självständig enhet.

De tidiga actor-modellerna var enligt Garnock-Jones (2016) enhetliga actor-modeller, med vilket han menade att allting skulle vara en actor. Han jämför dem med renodlad objektorienterad programmering där allting är objekt. Den strikta synen på actor-modeller menar Garnock-Jones luckrades upp i mitten av 1990-talet. Efter 1990-talet är det vanligare med modeller där actor-modellen är en del som läggs till i ett programmeringsspråk (Garnock-Jones, 2016). Fördelarna med en strikt actor-modell handlar till stor del om säkerhet medan det ofta finns praktiska skäl till att en programmerare går ifrån de strikta actor-språken (De Koster, Marr, Van Cutsem, & D'Hondt, 2015).

2.2 Påverkan från användandet av en actor-modell

Enligt Microsoft Dokumentation (2017b) kan en design med actorer bidra till programmeringslösningar för ett flertal olika typer av problem och situationer men det kan även begränsa utformningen. Det är alltså inte alltid är en optimal lösning att använda sig av en actor-modell. Några punkter som pekar på att en actor-modell kan hjälpa programmeringslösningen är enligt Microsoft Dokumentation (2017b) att problemet innefattar ett stort antal självständiga enheter av tillstånd och beräkningar, att utvecklaren önskar en lösning med enkeltrådade objekt som inte kräver betydande samarbete med externa komponenter samt att actorerna i lösningen inte innefattar oförutsedda I/O moment som kan blockera anrop (Microsoft Dokumentation, 2017b).

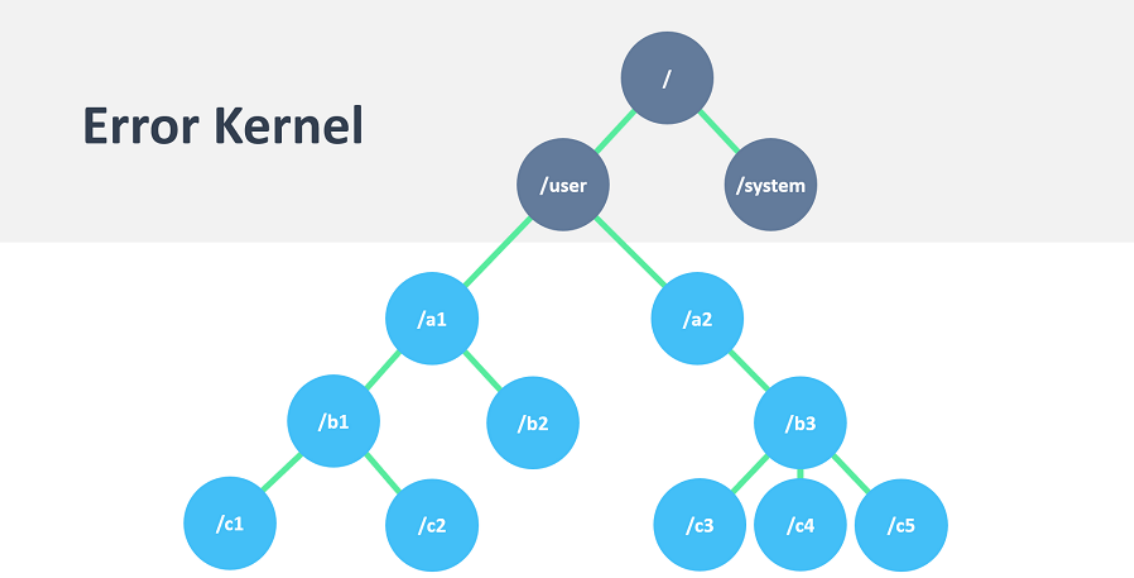
Genom att varje actor endast ska hantera en uppgift kan en kod skriven på actor-modellen antas bli mer renodlat. Det kan alltså bli enklare att överblicka vad en del av koden har som uppgift. Eftersom actorernas kommunikation endast sker genom

meddelanden till varandras brevlådor kan kommunikationen mellan actorerna ses som okomplicerad och lättfattlig. Enligt Akka.NETs dokumentation (2017d) kan en actor-modell tillåta ett system att bete sig mer på ett sätt som matchar en människas mentala modell över systemet (Akka.NET, 2017d). Alla actorer har alltså ett state som endast kan påverkas via meddelanden till actorn. Genom actorernas state kan kostsamma anrop till databaser även undvikas.

2.3 Olika Actor-modeller

2.3.1 Akka.NET

Enligt Akka.NETs dokumentation kan man se en programmeringslösning med actorer som en grupp personer där alla har fått deluppgifter. Varje funktion ska enligt Akka.NETs actor-struktur delas upp i så små delar att varje del bara innefattar en uppgift som en Akka.NET actor kan hantera. Deras olika funktioner kan tillsammans fungera som en organisationsstruktur. Den strukturen är i Akka.NET strikt heirarkisk, när en funktion behöver delas upp i delfunktioner sker det genom att actorn som hade ansvaret för funktionen skapar barn-actorer som får ansvaret för varsin delfunktion. Den ursprungliga actorn får ansvaret att övervaka dess barn-actorer under deras livstider. Nedan i figur 1 visas den hierarkiska strukturen hos Akka.NET actorer (Akka.NET, 2017a).



Figur 1. Akka.NET actorers hierarkiska struktur och felhanteringssystem (Akka.NET, 2017a).

Ovan, i figur 1, kan Akka.NET actorers hierarkiska struktur observeras. Eftersom alla Akka.NET actorer ingår i den hierarkiska strukturen så har alla actorer en övervakare, även kallad förälder. När en actor inte kan hantera sin uppgift eller när ett problem

uppstår skickas ett felmeddelande till actorns övervakare för att be den om hjälp. Kan inte övervakar-actorn hantera problemet skickar den ett felmeddelande till sin övervakare. Därigenom kan fel hanteras på rätt nivå i strukturen (Akka.NET, 2017a).

Det första som händer om något går fel i en actor är att den stängs av tillfälligt och ett meddelande skickas till förälder-actorn som är dess övervakare. Med att något går fel innebär att ett undantag har kastats eller att ett ohanterat undantag tas emot av actorn. Felinformationen tas emot av övervakaren som avgör hur felet ska hanteras. Standardåtgärden för en actor är att stänga ner sin barn-actorn och starta upp den igen (Akka.NET, 2017b).

Den översta actorn "/", i figur 1, kallas även rot-actorn och är att betrakta som föräldern och övervakaren till alla actorer i systemet eftersom det är den första actorn i system, den kommer även vara den sista actorn att sluta fungera när ett system stängs ner. Nästa lager i hierarkin "/user" och "system" hanterar två olika delar av actor-systemet. Actorn "/system" är övervakaren för själva systemet och interna Akka.NET actorerna. De actorer som tillhör och skapas i användarutrymmet övervakas och skapas av actorn "/user", detta är alltså föräldern till alla actorer som en utvecklare skapar. Dessa tre fördefinierade actorer kallas på engelska guardian, det vill säga en övervakare och beskyddare, eftersom de är systemets sista försvar - de får hantera alla ohanterade fel som uppstår längre ner och skickas vidare uppåt i hierarkin (Akka.NET, 2017b).

När ett Akka.NET actor-system ska designas krävs det eftertanke kring hur hierarkin ska se ut för att den ska fungera - vilken actor ska övervaka vilken. I Akka.NETs dokumentation menar de på att det inte finns en entydig ultimat lösning men att det finns ett antal regler som kan underlätta. Ett tecken på att en actor bör övervaka en annan är att den första hanterar den andres uppgift, exempelvis genom att skicka vidare deluppgifter till den. På det hela bör en actor övervaka de actorer som den har kunskap om hur fel bör hanteras i. En actor som innehåller viktig information eller data bör låta andra actorer längre ner i hierarkin hantera eventuella farliga uppgifter. Det ger en grundläggande regel att actorer bör designas på ett sätt där actorer skapar barn-actorer som får hantera funktioner som riskerar att leda till fel som kan skada actorn (Akka.NET, 2017a).

Actorer kommunicerar genom meddelanden som skickas till en annan actors brevlåda. Enligt Akka.NETs dokumentation (2017c) är det god sed att i förväg definiera vilka typer av meddelande som actorn kan ta emot. Dessa typer av meddelande kan definieras som klasser och sedan tas emot i actorns *OnReceive*-metod (Akka.NET, 2017c).

Akka.NETs dokumentation (2017a) menar på att flera actor-system kan vid behov samexistera inom Akka.NET och kommunicera med varandra, på det viset kan systemen utgöra byggstenar i ett större funktionellt system. Dock allokerar ett actor-system en eller ett flertal trådar och är därför en krävande struktur - vilket betyder att man endast bör skapa en per logisk applikation (Akka.NET, 2017a).

Enligt Akka.NETs dokumentation (2017a) är det i en del fall inte möjligt att undvika alla operationer som blockerar en tråd för andra operationer. I de fall blockeringar är oundvikliga rekommenderar Akka.NETs dokumentation fyra olika lösningar. Det första innebär att det blockerande anropet sker inom en actor designad med en trådpool som är tillräckligt stor eller helt ägnat för det aktuella anropet. Andra alternativet som Akka.NETs dokumentation föreslår är att det blockerande anropet sker inom en Task med en gräns för hur många anrop som tillåts ske. Det tredje anges vara att det blockerande anropet sker inom en Task som ger en trådpool med en övre gräns för antalet trådar som tillåts användas. Den fjärde lösningen Akka.NETs dokumentation ger för att hantera en blockerande operation är att tillägna en tråd till att hantera problematiken (Akka.NET, 2017a).

Konfigureringen av trådpooler är enligt Akka.NETs dokumentation (2017a) något de rekommenderar att programmeraren låter Akka.NET hantera. Det genom att den konfigureras i application.conf och instansieras i actor-systemet. I Akka.NETs dokumentation framgår även att meddelandehanteringen och hanteringen av resurser i actor-systemet inte är möjlig för programmeraren att kontrollera utan att detta sköts av Akka.NET (Akka.NET, 2017a).

Livscykeln för Akka.NET Actorer

Hierarkin för Akka.NETs actorer finns för att hantera livscykeln hos actorerna i actor-systemet. Livscykeln hos en actor startar när den skapas och avslutas när den på användarens begäran stoppas. När en actor stoppas kommer även alla dess barn-actorer att stoppas rekursivt. Därigenom undviks onödig allokering av resurser och läckor som öppna socklar (Akka.NET, 2017b).

En actor skapas med hjälp av konfigurationsklassen Props som används för att specificera premisserna för skapandet av actorer. Det finns flera sätt att skapa en instans av Props där olika argument kan användas för att definiera hur actorn ska skapas. Genom att använda en Props-instans som inparameter i den *ActorOf*-metod som finns tillgänglig från *ActorSystem* och *ActorContext* skapas en actor. När *ActorSystem*'s *ActorOf*-metod används skapas en actor på toppnivån, om *ActorContext* används kommer en barn-actor att skapas till den aktuella actorn (Akka.NET, 2017c).

Anropet till *ActorOf*-metoden skapar en instans av den angivna actorn och har ett returvärde som är en instans av *IActorRef*. Denna instans av *IActorRef* är det enda sättet att kommunicera med instansen av den korresponderande actorn. Instanserna av *IActorRef* och actorn har ett ett-till-ett förhållande, *IActorRef* instansen är oföränderlig (Akka.NET, 2017c).

Enligt Akka.NETs dokumentation (Akka.NET, 2017b) kan en actor stoppas både från insidan av actorn och utifrån. Det sker genom följande anrop *Context.Stop(self)* respektive *Context.Stop(actorRef)* där "*actorRef*" hänvisar till den actor som ska stoppas. Akka.NET rekommenderar att actorer stoppas från insidan med

Context.Stop(self) exempelvis genom ett fördefinierat stopmeddelande eller när actorn har slutfört sitt arbete (Akka.NET, 2017b).

I Akka.NET actorers API finns ett antal fördefinierade metoder som anropas vid olika tidpunkter av actorns livscykel. De kan överskuggas för att styra actorns beteende under livscykeln. De vanligaste av dessa metoder är *PreStart()* samt *PostStop()*, där den förstnämnda anropas efter att actorn startas men innan den hanterar sitt första meddelande och den sistnämnda anropas direkt efter att actorn stoppats (Akka.NET, 2017b). Som exempel skulle följande kodrad skriva ut "Actorn stoppad" i konsolfönstret efter att actorn hade stoppats:

```
protected override void PostStop() => Console.WriteLine("Actorn stoppad");
```

2.3.2 Service Fabric Reliable Actor

Inom Service Fabric finns två högnivåramverk för att bygga applikationer. Reliable Service APIs och Reliable Actor APIs (Bai, 2016). Enligt Microsoft Dokumentation (2017b) är Reliable Actors API baserat på Orleans Virtual Actor och är en programmeringsmodell som ger möjlighet att programmera enkeltrådat med skalbarhet och reliabilitetsäkerhet från Service Fabric (Microsoft Dokumentation, 2017b).

Varje actor i Service Fabric är enligt Microsoft Dokumentation (2017b) definierad som en instans av en actor-typ på samma vis som en .NET typ har olika instanser som är .NET objekt.

Enligt Sørensen (2015) är actorer ett bra angreppssätt inom Service Fabric när ett stort antal små objekt behöver kunna ha ett eget state. När en Actor service skapas inom Service Fabric läggs två projekt till, ett för actorer och ett för deras gränssnitt - interfaces. Det beror enligt Sørensen på att services inom Service Fabric inte refererar direkt till varandra utan går via deras interface. När en annan service interagerar med Actor servicen genom dess interface hanterar Service Fabric de faktiska actor-objekten (Sørensen, 2015).

Sørensen (2015) beskriver hur de funktioner som den korresponderande actorn kommer att ha definierats genom dess interface. Där anges hur actorns funktioner kommer se ut med deras in- och utparametrar. Interfacet ärver från klassen *IActor* och namnges motsvarande efter respektive actor, exempelvis om actor-klassen heter *NewActor* får interfacet namnet *INewActor* (Sørensen, 2015).

I sin introduktion till Service Fabric Reliable Actors skriver Sørensen (2015) om hur en referens till en specifik actor kan erhållas. Det sker genom ett *ActorProxy*-anrop med ett *ActorId* som inparameter. *ActorId* är vad som identifierar den specifika instansen av en actor-typ i Service Fabric. Vid ett *ActorProxy*-anrop där en icke-existerande actor-instans efterfrågas skapas den specifika actorn, om den däremot existerar skapas en referens till den instansen. Detta innebär att en actor-instans aldrig behöver skapas aktivt utan det sköts utav Service Fabric (Sørensen, 2015).

När själva actor-klassen skapas ärver den klassen *Actor* och implementerar sitt interface enligt Sørensens (2015) introduktion till Service Fabric Reliable Actors. Utöver de funktioner som definierats i actorns interface bör även metoden *OnActivareAsync* överskuggas. Den metoden körs när actorn skapas och kan användas för att skapa ett initialt state för actorn. I de fall state planeras att lagras i form av objekt eller klasser måste de serialiseras när klasserna definieras (Sørensen, 2015).

Enligt Microsoft Dokumentation (2017a) är *Data Contract* standarden för serialisering för Windows Communication Foundation (WCF). Alla primitiva datatyper i .NET ramverket går att betrakta som att de redan har *Data Contracts* och är klara att serialiseras. Nya typer eller klasser som skapas måste däremot ha ett *Data Contract* definierat innan de kan serialiseras. Det görs enligt Microsoft Dokumentation (2017a) genom att *[DataContract]* anges raden ovan klassdefinitionen och *[DataMember]* raden ovanför alla attribut i den aktuella klassen. Klassen *DataContractSerializer* serialiserade alla publika attribut och fält som inte aktivt är märkta att inte serialiseras (Microsoft Dokumentation, 2017a).

State-hantering för Service Fabric Reliable Actors

Tjänster som inte sparar sitt nuvarande tillstånd lokalt och inte behöver underhålla sitt nuvarande läge kallas för Stateless services eftersom de inte sparar sitt state lokalt. Tjänster som sparar sitt state lokalt kallas för Stateful services. Stateless services kan fortfarande ha värden men de gör de i ett externt datalager (Bai, 2016, s11).

En Stateful service kan leda till problem om den går ner, eftersom den då tar sitt state med sig vilket alltså kan leda reliabilitetsproblem och serviceavbrott för användaren. Det kan också bli problem med skalbarheten eftersom det, till skillnad från hos en Stateless service, spelar roll vilken instans av tjänsten som hanterar den specifika uppgiften. Det orsakas av tillståndet som tidigare diskuterats lagras lokalt i den instansen och för att uppnå konsistens i tjänsten (Bai, 2016, s11).

Reliable Actors är att betrakta som stateful services men behöver enligt Turecek och Delgado (2016) inte alltid spara sitt state på ett reliabilitetsäkert sätt. En utvecklare som använder sig av Reliable Actors kan välja, baserat på vilka datalagringskrav de har, på tre olika nivåer av state-lagring (Turecek & Delgado, 2016).

De tre nivåerna är enligt Turecek och Delgado (2016) “Beständigt state” (eng. persistent), “Flyktigt state” (eng. volatile) och “Ej beständigt state”. “Beständigt” innebär att state både kopieras till tre kopior av instansen och sparas på en hårddisk. Det är den säkraste nivån av lagring av state och är användbar om applikationen inte får riskera att instansen tappar sitt state. Använder en utvecklare sig av beständig state-lagring kan instansen behålla sitt state även om klustret skulle få ett avbrott. Även för nivån “flyktigt” kopieras state till tre kopior av instansen men på den nivån sparas det endast i minnet. Det innebär att instansen är motståndskraftig mot nod- och actor-krascher samt under uppdateringar men att state skulle förloras om de tre kopiorna av

instansen skulle förloras under ett avbrott. Den sista nivån “ej beständigt” innebär att instanserna inte kopieras och state inte heller lagras på hårddisken. I de fall utvecklaren inte behöver ha en säker lagring av state hos sina actorer kan denna nivå användas och minne sparas (Turecek & Delgado, 2016).

I början av varje actor kan utvecklaren ange vilken nivå lagringen av state ska vara på i den actorn. För att ange nivån i C# inleds koden till actorn med någon av de tre följande raderna:

```
[StatePersistence(StatePersistence.Persisted)]
[StatePersistence(StatePersistence.Volatile)]
[StatePersistence(StatePersistence.None)]
```

Första raden skulle alltså ge actorn en beständig lagring av state, andra en flyktig och tredje en ej beständig lagring (Turecek & Delgado, 2016).

Registrera actorn

För att kunna använda en Service Fabric Actor från andra services måste den enligt Sørensen (2015) registreras hos Service Fabric. Det görs enligt Sørensens introduktion till Service Fabric Actors i main-funktionen för actor-servicen med hjälp av följande kodrad:

```
ActorRuntime.RegisterActorAsync<xxActor>(
(context, actorType) => new ActorService(context, actorType)).GetAwaiter().GetResult();
```

Detta bör göras på samma vis för alla typer av actorer som kommer att ingå i actor-servicen (Sørensen, 2015).

3. Metod

För att undersöka vilken av de två valda modellerna, Akka.NET och Service Fabric Reliable Actor, som kan anses passa bättre ihop med Caspecos lönemotor gjordes en simulerad lönekörning i vardera actor-modell. Därefter gjordes en implementation av den valda modellen i delar av lönesystemet.

3.1 Simulering med Actor-modellen Akka.NET

Simuleringen av en lönekörning med actor-modellen Akka.NET byggdes upp av nio actorer. Hierarkin för actorerna i simuleringen och actorernas namn kan ses nedan i figur 2.

Figur 2 Hierarki Simulering med actor-modellen Akka.NET

Ovan, i figur 2, kan hierarkin för simuleringen observeras. Den första actorn som är högst i hierarkin, *ReportGenActor*, startar processen för insamlandet av information som krävs för att sätta igång en lönekörning genom att skicka ett startmeddelande till sitt barn *InfoActor*. *InfoActor* skapar i sin tur barn-actorer, *EmployeeActor*, *CustomerActor*, *WorkingShiftActor*. De actorerna simulerar en beräkning genom att pausa sin tråd i 3 s innan den skickar ett meddelande tillbaka till den som gav dem ett startmeddelande. *InfoActor* skapar även en aggregator-actor *AggActor*, den har till uppgift att kontrollera en lista med actor-objekt och meddela *InfoActor* när alla actorer i den är klara. När *InfoActor* får meddelandet av *AggActor* att alla informationshämtande actorer är klara skickar den ett meddelande till sin actor-förälder *ReportGenActor* om att informationen är hämtad.

När *ReportGenActor* får meddelandet från *InfoActor* att all information som krävs för lönekörningen är inhämtad startar den sitt andra barn *PayrollActorCoordinator*.

PayrollActorCoordinator skapar även den en aggregator-actor. Den skapar även en *PayrollActor* för varje anställd, dessa läggs i en lista som *AggActor*-instansen använder för att kontrollera när alla instanser är klara med sina uppgifter. Varje *PayrollActor* pausar sin tråd i 1 s för att simulera beräkning samt skapar i sin tur en *PayrollMonthActor* för efterfrågad månad, som vid behov kan skapa *PayrollMonthActor* för tidigare månader. I simuleringen pausar *PayrollMonthActor* sin tråd i 1 s för att simulera beräkning innan den meddelar sin förälder att den är klar. När *PayrollActor* fått informationen om att beräkningarna är klara meddelar den sin förälder *PayrollActorCoordinator* att den är klar. När alla *PayrollActor* i listan *AggActor* använder sig av har meddelat att de är klara är lönekörningssimuleringen är klar.

För att inte blockera trådar i onödan använder simuleringen Tell-funktionen istället för Ask-funktionen som låser actorn i väntan på svar. Genom att simuleringen använder sig av aggregator-actorer kan avsändar-actorn få veta när alla de igångsatta actorerna är klara utan att använda sig av Ask-funktionen.

3.2 Simulering med actor-modellen Service Fabric Reliable Actor

I filen *program,* för projektet *Actors* i simuleringen med actor-modellen Service Fabric Reliable Actor, registreras alla actorer. De actorer som skapas för simuleringen är *EmployeeActor*, *PayrollActor*, *PaylistActor*, *LockActor*, *WorkingShiftsActor*, *InstancesActor* samt *SystemSettingsActor*. De identifieras vid behov av olika nycklar i form av strängar som lagras i instansvariabler. Actorerna har state-nivån ”beständig” och använder sig av *StateManager* för att hantera sina state.

Actorn *PayListActor* identifierades av företagets namn och det efterfrågade lönedatumet. I *PayListActor* skapades en *EmployeeActor* som vid efterfrågan gav en simulerad lista av anställda. *EmployeeActor*’n har en nyckel som består av namnet på företaget den ska simulera att hantera anställda för, vilket simuleras i form av listor med heltal. För varje heltal i den listan skapade *PayListActor* en *PayrollActor* som identifierades av företagsnamnet, lönedatumet och anställningsnumret som kombinerat nyckel. I *PayListActor* efterfrågades löneberäkningar från alla *PayrollActor*’s som simulerade en beräkning genom en förinställd fördröjning. Genom en *await Task.WhenAll(tasks)* pågick dessa simulerade beräkningar parallellt och inväntades innan det simulerade resultatet sammanställdes. I simuleringen med Service Fabric Reliable Actor valdes alltså denna lösning istället för användandet av aggregator-actorer.

Övriga actorer i denna simulering hade som beteende att ge ett returvärde som simulerade ett värde eller inställning. De hade inga specificerade nycklar som identifierade dem utan allmänna *actorId*’s skapades och de förväntades ge samma returvärde för alla instanser.

3.3 Skillnader mellan Akka.NET och Service Fabric Reliable Actor

I Akka.NET är State lagrat som en vanlig variabel jämfört med den *Statefulness* som finns inbyggt i Service Fabric. Det går att komma runt hur Akka.NET lagrar *State* genom Akka.NET *Persistence* men den inbyggda lösningen på hur State lagras i Service Fabric är gjord för att användas för att göra State persistent och automatkopierad.

Den största skillnaden som upptäcktes mellan de två actor-modellerna var att Akka.NET förespråkade en striktare hierarki och krävde en mer strikt actor-modell. En stor skillnad var även att Service Fabric Reliable Actor är ett ramverk från Microsoft Azure, vilket är av vikt då Caspecos lönemotor är uppbyggt på Microsoft Azure i dagsläget.

3.4 Implementation av Proof-Of-Concept

För att kunna utveckla en applikation med hjälp av Service Fabric krävs en utvecklingsmiljö och ett Service Fabric kluster (Bai, 2016, s11). Att sätta upp en utvecklingsmiljö kräver Visual Studio och Service Fabric SDK. Genom att installera Service Fabric SDK kan ett lokalt Service Fabric kluster med flera noder skapas. På det lokala klustret kan applikationen testköras under utvecklingen av applikationen. När en applikation ska lanseras rekommenderas ett kluster på en extern server men för utveckling av ett Proof-Of-Concept fungerar ett lokalt kluster (Bai, 2016, s12). Under utvecklingen av detta Proof-Of-Concept kommer därför ett lokalt Service Fabric kluster att användas genom Service Fabric SDK. Det lokala klustret kan hanteras exempelvis genom Service Fabric Local Cluster Manager (Bai, 2016, s22-26).

Implementationen av Service Fabric Reliable Actors in i lönemotorn skedde efter den tidigare beskrivna simuleringen av en lönekörning helt fristående från det nuvarande lönesystemet. I simuleringen ingick actorer som planerades att ingå i det slutliga proof-of-concept men innehållet i actorerna var en förenklad pseudo-kod. Därigenom skulle ett samspel mellan de relevanta actorerna uppnås utan att den existerande, ganska invecklade, koden för lönemotorn behövde användas. Implementationen av actorer som hjälpmedel i lönemotorn byggde på att låta olika actorer hantera delar av koden. De delar som valdes ut att implementeras i ett första steg var de som ansågs kunna separeras från övriga delar på ett tydligt sätt. Till skillnad från simuleringarna av lönekörningar på actor-modell som gjordes i de två olika utvalda actor-modellerna var actorerna i den slutliga implementationen endast inslag i den vanliga koden. Strukturen för hur koden var uppbyggd var beroende av hur lönekörningsmotorn var uppbyggd från början. Eftersom den befintliga koden för lönemotorn var väldigt omfattande, komplex och delvis snårig gjordes många avvägningar var de olika actorerna skulle anropas. Dessa avvägningar påverkade hur den aktuella actorn skulle innehålla och anropas. Genom detta påverkade den befintliga koden utformningen av implementationen.

Under implementationen av Service Fabric Reliable Actor in i lönemotorn gjordes avgränsningen att implementera get-funktioner för actorerna. Anledningen till den avgränsningen var att den ökade komplexiteten i att använda set-funktioner mot databasen ej skulle hjälpa syftet med undersökningen i förhållande till den utökade omfattningen av projektet det skulle leda till.

3.4.1 Tillvägagångssätt

Inför implementationen av ett actor-system identifierades vilka typer av actorer som ansågs nödvändiga. Det gjordes genom att olika roller inom lönesystemet som skulle kunna göras om till actorer plockades ut. De roller som ansågs lämpliga att göra om till actorer blev *EmployeesActors, PayrollActor, InstancesActor, LockActor* och *WorkingShiftsActor*. Därefter togs beslut om exakt vad varje actor skulle innefatta och vad som skulle vara kvar i den övriga koden. Som tidigare nämnts anpassades det efter hur den befintliga koden för de olika funktionerna var uppbyggt och med fokus på olika get-funktioner.

Viktiga avvägningar som gjorts under projektets gång och implementationen av ett proof-of-concept är vad varje actor ska innehålla och göra. Den logiska struktur som togs fram i början av projektet och delvis användes i de simuleringar som gjordes i två olika actor-modeller användes inte fullt ut i implementationen av Service Fabric Reliable Actors i lönemotorn. Detta berodde på att anpassningar var tvungna att ske till lönemotorns struktur och upplägg. De actorer som implementerades i lönemotorn valdes ut av de som använts i simuleringen och vars motsvarande kod i lönemotorn bedömdes gå att omvandla. Urvalet av actorer byggde alltså mycket på hur den befintliga koden var uppbyggd. Orsaken till det var delvis att den befintliga koden till lönemotorn är väldigt snårig och komplex. Detta ledde alltså till att implementationen anpassades en del till lönemotorns befintliga kod istället för att koden helt anpassades efter actor-modellen.

En viktig aspekt av implementationen var att ingen information i actorerna bedömdes kräva en högre nivå av state-lagring. Eftersom all information enkelt kunde hämtas upp ur databasen igen fanns inte behovet av att lagra den som state på ett säkert sätt i actorn. Ifall actorn på något sätt skulle krascha var informationen alltså lättillgänglig igen. Den extra tid det skulle ta att åter hämta informationen bedömdes vara att föredra framför att allokera extra minne. Därför lagrades den hämtade informationen i instansvariabler i form av listor med objekt och actorerna hade state-nivån ”Ej beständig”. Till skillnad från i simuleringen användes här alltså inte *StateManager*’n.

En av de avvägningar som gjordes i början av implementationen var om *EmployeesActor* skulle hantera all information om alla anställda i det aktuella företaget eller om varje actor skulle hantera informationen för en enda anställd och flera *EmployeeActors* alltså skulle vara aktiveras och vara involverade i varje löneanrop. Där föll valet på att hämta en lista med anställd i en *EmployeesActor*. Det byggde på hur koden för lönemotorn såg ut och att den ofta frågade efter alla anställda.

Innan actorerna i lönemotorn skapades och implementerades in i den befintliga koden gjordes först en *NewsActor*. Den hade i uppgift att hämta nyheterna till startsidan på ett förvalt språk. Denna actor skapades för att nyhetshämtningen var en avgränsad del i den befintliga koden som ansågs lämplig att använda för att skapa en enkel actor. Genom att inleda med denna actor kunde samspelet med en Service Fabric Reliable Actor och den befintliga koden testas innan den mer komplicerade lönemotorn skulle användas. Denna *NewsActor* har även använts för att på ett tydligt sätt studera hur själva koden förändras när en Service Fabric Reliable Actor används.

3.4.2 ActorId klasser

För att kunna aktivera rätt actor-instans på ett enkelt sätt skapades för varje actor-typ en klass som hanterade nycklarna för actorn. Dessa klasser lades i *namespace MarcActor.Interfaces* i en fil tillsammans med respektive actor-typ. Ett exempel på hur dessa *ActorId* klasser är uppbyggda kan observeras nedan genom *PayrollActors ActorId*-klass:

```
public class PayrollActorId
{
    public string SystemName;
    public long EmployeeId;
    public Date PayDate;

    public PayrollActorId(string systemName, long employeeId, Date payDate)
    {
        SystemName = systemName;
        EmployeeId = employeeId;
        PayDate = payDate;
    }

    public override string ToString()
    {
        return $"{SystemName};{EmployeeId};{PayDate}";
    }

    public ActorId Id
    {
        get
        {
            return new ActorId(ToString());
        }
    }

    public static PayrollActorId Parse(ActorId id)
    {
        var parts = id.GetStringId().Split(';');
        return new PayrollActorId(parts[0], long.Parse(parts[1]), Date.Parse(parts[2]));
```

```
    }
}
```

Klassen *PayrollActorId* har tre instansvariabler som används som nycklar för en specifik instans av actorn *PayrollActor*, nämligen *SystemName*, *EmployeeId* och *PayDate*. I konstruktorn anges dessa instansvariabler till de inparametrar som konstruktorn anropats med. Den andra funktionen *ToString()* har inga inparametrar och har ett returvärde som består av de tre första instansvariablerna formaterat som en sträng. Den fjärde instansvariabeln *Id* har en *get* med returvärdet ett *ActorId* skapat med de övriga instansvariablerna när de är formaterade till en sträng. Sist i klassen finns en funktion för att returnera ett *PayrollActorId* från ett *ActorId*, vilket sker genom att strängen som identifierar *ActorId*:t parsas till de tre nycklarna som identifierar ett *PayrollActorId*.

Alla actorer i projektet har motsvarande klasser för att hantera sina nycklar och *ActorId*:n. Den sista funktionen, *Parse(ActorId id)*, finns för att underlätta att komma åt nycklarna till actorn samt att underlätta i programmeringen och undvika fel som skulle kunna uppstå ifall nycklarna i actorn skulle hämtas från en sträng.

Nycklar actorer

Nyckeln för *NewsActor* är *Language* vilket motsvarar vilket språk som nyheterna ska hämtas på. Det innebär att det endast finns en actor-instans för ett specifikt språk där alla nyheter på det språket hämtas och lagras.

För *EmployeesActor* är nyckeln *SystemName*, det innebär att nyckeln till actor-instansen är det aktuella företaget. En avvägning mellan att ha en actor med både företaget och ett ID för den anställde som nyckel och endast företaget som nyckel gjordes. Valet föll på företaget som nyckel eftersom många operationer i koden sker med alla anställda i ett företag. Även inför skapandet av *InstancesActor* gjordes ett val där ena alternativet var en kombinerad nyckel uppbyggd av företaget och ett ID för den anställda och det andra en nyckel som endast bestod av företaget. På liknande vis som för actorn för anställda valdes endast företaget eftersom de aktuella anropen ofta efterfrågar *PayrollArticleInstance* för alla anställda samtidigt. Med samma logik valdes endast företaget som nyckel för *WorkingShiftsActor*.

Övriga actorers nycklar valdes utan avvägningar eftersom deras uppgifter var mer självskrivna. För actorn *StationsActor* blev nyckeln det aktuella företaget genom att actorn hade i uppgift att hämta de för ett företag aktuella stationerna. Actorn som planeras hantera låsta datum för en lönekörning, *LockActor*, har även den företaget som nyckel. Där var valet givet genom att de låsta datumen avgörs av just vilket företag det handlar om. Den sista actorn, *PayrollActor*, hade en kombination av företaget, ett ID för den aktuelle anställde och lönedatumet. De tre skapar tillsammans en unik lönekörning och är därför vald som nyckel till *PayrollActor* som skapades för att hantera en unik lönekörning.

3.5 Utvärdering

Slutligen utvärderades projektet med avseende på om koden har kunnat förenklats samt på hur prestandan har förändrats. Förenklingen av koden har utvärderats genom jämförelser av koden för hämning av nyheter innan och efter Service Fabric Reliable Actor'n *NewsActor* infördes.

Förändringen av prestandan har utvärderats genom upprepade tidsmätningar av olika anrop innan och efter införandet av Service Fabric Reliable actorerna *EmployeesActor*, *LockActor*, *StationsActor* och *WorkingShiftsActor*. Dessa actorer valdes ut med grund i att de motsvarande anropen var möjliga att göra avskilda och därigenom mäta med avseende på tiden, ett och ett.

Tidsmätningarna gjordes med hjälp av en *Stopwatch Class* beskriven i Microsoft dokumentation (2017c). Koden byggde på Microsoft dokumentations exempel för hur *Stopwatch Class* kan användas för att beräkna prestandadata. Fullständig kod för tidsmätningarna återfinns i Appendix A.

4. Resultat

4.1 Förenklad modell av kod

En del av syftet med projektet var att förenkla förståelsen av koden till löneanropsmotorn. Ett exempel på hur Service Fabric Reliable Actor har kunnat förändra förståelsen av koden i detta projekt är koden som hanterar hämtningen av nyheter till startsidan. Nedan kommer först koden utan actorer presenteras och därefter kommer actorn som tagit över uppgiften i koden att introduceras.

4.1.1 Anrop efter nyheter utan Service Fabric Reliable Actor

Anrop efter nyheter sker i programmet i koden för *NewsService* som ursprungligen ser ut som följer:

NewsService:

```
namespace Api.Logic.Services.Common
{
   public interface INewsService
   {
       Task<IEnumerable<NewsService.NewsRecord>> GetAsync(string[] categories, string language);
   }
   public class NewsService : INewsService
   {
       private const string baseUrl = @"http://www.caspeco.se";
       public class NewsContainer
       {
           public string status;
           public int count;
           public int count_total;
           public int pages;
           public NewsRecord[] posts;
       }
       public class Author
       {
           public string name;
           public string first_name;
           public string last_name;
       }
       public class Category
       {
           public int id;
           public string slug;
           public string title;
       }
```

```
        public class NewsRecord
        {
            public int id;
            public string status;
            public string slug;
            public string title;
            public string excerpt;
            public DateTime date;
            public DateTime modified;
            public Category[] categories;
            public Author author;
            public string content;
        }
        public async Task<IEnumerable<NewsRecord>> GetAsync(string[] categories, string language)
        {
            HttpResponseMessage response;
            var result = new List<NewsRecord>();
            // The categories can have language suffix
            List<string> allCategories = new List<string>(categories);
            foreach (string cat in categories)
            {
                allCategories.Add($"{cat} @{language}");
            }
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri(baseUrl);
                response = await client.GetAsync($"/?json=get_recent_posts&lang={language}");
                response.EnsureSuccessStatusCode();
                string responseBody = await response.Content.ReadAsStringAsync();
                var data = JsonConvert.DeserializeObject<NewsContainer>(responseBody);
                foreach (NewsRecord rec in data.posts)
                {
                    foreach (Category category in rec.categories)
                    {
                        if (allCategories.Contains(category.title))
                        {
                            result.Add(rec);
                            break;
                        }
                    }
                }
            };
            return result;
        }
    }
}
```

4.1.2 Anrop efter nyheter med Service Fabric Reliable Actor

Den actor som ansvarar för hämtning av nyheter hänvisas i nedanstående kod och hädanefter som *NewsActor*. *NewsActor* är som resterande actorer i detta projekt uppdelat i två, actorn och dess interface - hädanefter hänvisat som och namngiven till *INewsActor*. De anropas i koden för *NewsService* med Service Fabric Reliable Actor som ser ut som följande:

NewsService:

```
namespace Api.Logic.Services.Common
{
    public interface INewsService
    {
       Task<IEnumerable<NewsRecord>> GetAsync(string[] categories, string language);
    }
    public class NewsService : INewsService
    {
        public async Task<IEnumerable<NewsRecord>> GetAsync(string[] categories, string
language)
        {
            var newsActor = ActorProxy.Create<INewsActor>(new
NewsActorId(language).GetActorId());
            return await newsActor.GetNews(categories);
        }
    }
}
```

Nedan observeras koden till *INewsActor* med förklaringar. Vissa förkortningar har gjorts för att underlätta överblicken. Hela koden återfinns i appendix.

INewsActor:

```
namespace MarcActors.Interfaces
{
    [DataContract] //Serialisering av klassen NewsContainer vilket krävs eftersom
    public class NewsContainer //NewsActor kommer att skicka NewsContainers
    {
        [DataMember] //Serialisering varje attribut i klassen NewsContainer
        public string status;
        [DataMember]
        public int count;
        [DataMember]
        public int count_total;
        [DataMember]
        public int pages;
        [DataMember]
        public NewsRecord[] posts;
```

```
    }
[... Som för NewsContainer introduceras och serialiseras: Author, Category och NewsRecord. …]
    public class NewsActorId //Här skapas en klass som identifierar en instans av NewsActor
    {
        public string Language; //Det görs med nyckeln Language

        public NewsActorId(string language) //Konstruktor för NewsActorId
        {
            Language = language;
        }

        public ActorId GetActorId() //Metod för att hämta ett NewsActorId
        {
            return new ActorId(Language);
        }

        public static NewsActorId Parse(ActorId actorId)
        {
            return new NewsActorId(actorId.GetStringId());
        }
    }
    public interface INewsActor : IActor //Interfacet för NewsActor som ärver IActor
    {
        Task<List<NewsRecord>> GetNews(string[] categories); //Här anges alla metoder för
NewsActor
    }
}
```

Nedan följer koden för själva *NewsActor* med förklaringar.

NewsActor:

```
namespace MarcActors
{
    [StatePersistence(StatePersistence.None)] //State varken kopieras eller sparas till hårddisk
    internal class NewsActor : Actor, INewsActor //Ärver Actor och sitt interface
    {
        private const string baseUrl = @"http://www.caspeco.se";
        private NewsActorId _actorId; //Har ett actorId som definierats i INewsActor
        private NewsContainer _newsRecords;
        private IActorTimer _clearCacheTimer;

        public NewsActor(ActorService actorService, ActorId actorId) //Konstruktor för NewsActor
            : base(actorService, actorId)
        {
            _actorId = NewsActorId.Parse(actorId); //ActorId anges via metod som definierats i
INewsActor
        }
```

```
        protected override Task OnActivateAsync() //Här anges vad som sker när en instans
aktiveras
        {
            _clearCacheTimer = RegisterTimer(
                ClearCache,                    //Återanrop till metoden ClearCache
                null,
                TimeSpan.FromMinutes(10),  //Minuter innan återanropet aktiveras
                TimeSpan.FromMinutes(10)); //Tid mellan upprepningar av återanrop

            return base.OnActivateAsync();
        }
        private async Task ClearCache(object arg) //Metod som rensar sparade nyheter
        {
            _newsRecords = null;
}
        protected override Task OnDeactivateAsync() //Vid avaktivering av actorn avregistreras
timern
        {
            if (_clearCacheTimer != null)
            {
                UnregisterTimer(_clearCacheTimer);
            }

            return base.OnDeactivateAsync();
        }
        private async Task<NewsContainer> ReadNewsRecords(string language) //Metod som hämtar
nyhet
        {
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri(baseUrl);
                var response = await
client.GetAsync($"/?json=get_recent_posts&lang={language}");
                response.EnsureSuccessStatusCode();
                string responseBody = await response.Content.ReadAsStringAsync();
                var data = JsonConvert.DeserializeObject<NewsContainer>(responseBody);
                return data;
            };
        }
//Nedan följer den metod som definierades i INewsActor att NewsActor skulle innehålla. Dess
resultat med alla nyheter som hämtats in i NewsActor kommer att returneras via INewsActor.
        public async Task<List<NewsRecord>> GetNews(string[] categories)
        {
            NewsContainer news;
            if (_newsRecords == null)
            {
                _newsRecords = await ReadNewsRecords(_actorId.Language);
            }
            if (_newsRecords.count == 0 || _newsRecords.posts == null)
```

```
            {
                return new List<NewsRecord>();
            }
            var result = _newsRecords.posts.Where(post => post.categories.Any(category =>
categories.Any(queryCategory => queryCategory == category.slug))).ToList();
            ActorEventSource.Current.ActorMessage(this, $"Returning {result.Count()} posts");
            return result;
        }
    }
}
```

4.1.3 Utvärdering av förenkling av kod

Teorin var att genom att isolera ett moment i en actor skulle en modell av hur en kod är uppbyggd kunna förenklas. För att testa detta har *NewsActor,* som skulle hantera hämtandet av nyheter, studerats. De olika koderna för anrop efter nyheter – med och utan Service Fabric Reliable Actorn *NewsActor* – återfinnes i underavsnitten 4.1.1 och 4.1.2 ovan.

Den del av den befintliga koden som hämtar nyheter kallas *NewsService* och innehåller ett antal underklasser som definierar de olika objekt som krävs för att hämta in nyheterna. Den innehåller även en funktion som från ett antal kategorier och ett språk hämtar in alla relevanta nyheter från en angiven adress. Med Service Fabric Reliable Actor *NewsActor* implementerat förkortas koden för *NewsService* markant till att endast skapa en *NewsActor* och från den be om nyheterna.

I actorns interface *INewsActor* definieras och serialiseras de objekt som kommer att skickas från *NewsActor*'n. Det definieras även ett *NewsActorId* på det sätt som beskrivits i metodavsnittet. *NewsActor* innehåller utöver funktioner för att hämta in nyheterna även en timer som raderar nyheter efter tio minuter. Den timern finns för att låta nyheterna korttidslagras i *NewsActor*. Därigenom behöver inte nyheterna hämtas varje gång startsidan uppdateras utan att gamla nyheter kommer att visas.

Med en Service Fabric Reliable Actor implementerad blev enligt denna jämförelse koden längre. Varje del blev dock mer renodlad i sin funktion. *NewsService* innehöll endast en funktion som frågade efter nyheter. Actorns interface definierade *ActorId*'t för actorn, vad som actorn skulle kunna skicka samt vilka funktioner den skulle innehålla. Själva actorn innehöll två moment, hämtningen av nyheter den var till för samt en extratillagd timer och förmågan att spara nyheter. Timern och förmågan att spara nyheter innefattades tidigare ej av koden utan var en utnyttjning av actorns förmåga att spara nyheterna som en instansvariabel för att ge snabbare svarstid på kommande anrop.

4.2 Prestanda, tidsmätning

För att mäta vilken prestandaförändring en övergång till en actor-modell kan ge har upprepande anrop genomförts, med den gamla lösningen och med hjälp av Service Fabric Reliable Actors.

De fyra testen av olika anrop har summerats i tabeller i respektive anropsavsnitt. De olika värdena för första anropet, genomsnittliga anropet samt för variationsbredden har betraktats som diskreta stokastiska variabler.

Väntevärdet har beräknats med följande formel $\mu = \sum_x xP(x)$ där x är utfallet och P(x) är sannolikheten för utfallet, vilket i dessa fall innebär att väntevärdet kommer att uppskattas till medelvärdet för stickprovet som består av de fyra testen för de olika anropen.

Variansen har beräknats med följande formel $\sigma^2 = \sum_{i=1}^{4} P(x_i)(x_i - \mu)^2$ där μ är det beräknade väntevärdet och x_i varje de olika utfallen.

4.2.1 Mätning av anrop efter arbetspass

Tabell 1 – Statistik för testerna av anrop efter arbetspass utan respektive med actor

Anrop efter arbetspass	**Utan actor**	**Med actor**	**Procentuell skillnad**
Väntevärde första anropet	6306,5 ms	7650,8 ms	+21,3%
Varians första anropet	907 229,3 ms^2	512 750,2 ms^2	-43,5%
Väntevärde genomsnittliga tiden bortsett från första anropet	19,7 ms	18,9 ms	-4,1%
Varians genomsnittliga tiden bortsett från första anropet	0,7 ms^2	0,06 ms^2	-91,4%
Väntevärde variationsbredd utan första anropet	52,6 ms	1394,0 ms	+2550,2%
Väntevärde variationsbredd utan första anropet, utliggare borttagen	52,6 ms	47,3 ms	-10,1%
Varians variationsbredd utan första anropet	218,1 ms^2	5 313 698,0 ms^2	+2436258,6%
Varians variationsbredd utan första anropet, utliggare borttagen	218,1 ms^2	532,0 ms^2	+144,0%

Det går att i tabell 1 observera att första anropet efter arbetspass med hjälp av en Service Fabric Reliable Actor tog i snitt 21,3% längre tid än utan en Service Fabric Reliable Actor. I tabell 1 syns även att det varierade mindre mellan testerna hur snabbt första anropet tog. Väntevärdet för den genomsnittliga tiden minskade med 4,1 % genom användandet av en actor, det går även att se att variansen på den genomsnittliga tiden minskade med 91,4%. Variationsbredden, efter en utliggare togs ur testet, minskade med 10,1% vid användandet av en actor. Det går däremot även att se att variansen på variationsbredden gick upp med 144%.

Testen av anropen efter arbetspass tyder på att för detta anrop tar första gången detta anrop körs 1,3 sekunder längre med en actor än utan. För följande gånger anropet

kördes tog det däremot 0,8 ms kortare tid. Det innebär alltså att användandet av Service Fabric Reliable Actor för detta anrop skulle gå jämnt ut på 1680 anrop. För att börja spara tid totalt sett skulle det alltså krävas närmare 1700 anrop.

Variationen hos anrop som gjordes med hjälp av Service Fabric Reliable Actor var lägre än hos de som gjordes utan hjälp av en actor för de första anropen och genomsnittet för de följande anropen. Det innebär alltså att det skulle kunna vara mer förutsägbart hur lång tid de första anropen och genomsnittet för följande anropen kan komma att vara med en actor. Däremot tyder datan på att tiden kan variera mer mellan de följande anropen med en Service Fabric Reliable Actor och varje anrop efter den första vara mer oförutsägbar, det eftersom variationsbredden både är större och mer varierande i datan med en actor.

4.2.2 Mätning av anrop efter anställda

Tabell 2 - Statistik för testerna av anrop efter anställda utan respektive med actor

Anrop efter anställda	**Utan actor**	**Med actor**	**Procentuell skillnad**
Väntevärde första anropet	5438,1 ms	13 930,6 ms	+156,2%
Varians första anropet	1 964 884,0 ms^2	17 425 559 ms^2	+786,8%
Väntevärde genomsnittliga tiden bortsett från första anropet	25,25 ms	20,0 ms	-20,8%
Varians genomsnittliga tiden bortsett från första anropet	0,24 ms^2	2,3 ms^2	+858,3%
Väntevärde variationsbredd utan första anropet	282,0 ms	112,6 ms	-60,1%
Varians variationsbredd utan första anropet	5654,8 ms^2	2543,6 ms^2	-55,0%

Första anropet för att få tag på information om anställda tog 156,2% längre tid när en actor användes. Variansen för anropet tog nästan 9 gånger så lång tid. Väntevärdet för de kommande anropen minskade däremot med 20,8% när en actor användes. Variationsbredden på de senare anropen gick ner med 60,1% när en actor användes.

För de inledande anropen i varje test tyder datan på att det både tar längre tid i genomsnitt och varierar mer hur lång tid första anropet tar om man använder en actor för detta anrop. Däremot för de kommande anropen tog det generellt kortare tid för

testerna med en Service Fabric Reliable Actor, variansen för genomsnittet för de på den första anropet följande anropen var även den lägre för testerna med en actor. För anrop som frågar efter anställda skulle det enligt värdena i tabell 2 ta nästan 1620 anrop för att användandet av en Service Fabric Reliable Actor skulle gå jämnt ut.

4.2.3 Mätning av anrop efter låsta datum

Tabell 3 - Statistik för testerna av anrop efter låsta datum utan respektive med actor

Anrop efter låsta datum	**Utan actor**	**Med actor**	**Procentuell skillnad**
Väntevärde första anropet	8700,0 ms	2482,6 ms	-71,5%
Varians första anropet	25 596 397 ms^2	727 227,4 ms^2	-97,2%
Väntevärde genomsnittliga tiden bortsett från första anropet	15,7 ms	13,9 ms	-11,5%
Varians genomsnittliga tiden bortsett från första anropet	0,16 ms^2	0,005 ms^2	-96,9%
Väntevärde variationsbredd utan första anropet	255,0 ms	108,8 ms	-57,3%
Varians variationsbredd utan första anropet	27 657,9 ms^2	1479,5 ms^2	-94,7%

I tabell 3 går det att observera att första anropet efter låsta datum gick 71,5% snabbare med hjälp av en Service Fabric Reliable Actor och variansen sjönk med 97,2%. Det gick alltså snabbare med en actor än utan i motsats till testerna av de andra typerna av anrop. Den genomsnittliga tiden sjönk med 11,5% och variansen med 96,9% när en actor användes i anropen efter låsta datum.

Detta innebär alltså att datan för anrop efter låsta datum med, respektive utan, en Service Fabric Reliable Actor tyder på att både första anropet och de följande anropen gick snabbare med en actor än utan. De visade även på att anropen varierade mindre med avseende på hur lång tid de tog.

4.2.4 Mätning av anrop efter stationer

Tabell 4 - Statistik för testerna av anrop efter stationer utan actor

Anrop efter stationer	**Utan actor**	**Med actor**	**Procentuell skillnad**
Väntevärde första anropet	71,2 ms	4477,6 ms	6188,8%
Varians första anropet	288,1 ms^2	939 655 ms^2	326055,8%
Väntevärde genomsnittliga tiden bortsett från första anropet	10,8 ms	15 ms	38,9%
Varians genomsnittliga tiden bortsett från första anropet	0,007 ms^2	0,06 ms^2	757,1%
Väntevärde variationsbredd utan första anropet	30,1 ms	158,5 ms	426,6%
Varians variationsbredd utan första anropet	3,6 ms^2	5146,8 ms^2	142866,7%

Första anropet efter stationer tog med en Service Fabric Reliable Actor 6188,8% längre tid än utan en actor och det går i tabell 4 att se att även variansen för första anropet ökade extremt mycket. I tabell 4 går det att uttyda att även de följande anropen tog längre tid, dock markant mindre ökning än för första anropen.

Datan från tabell 4 visar alltså på att testerna utan Service Fabric Reliable Actor varit det snabbare alternativet både för första och för de följande anropen efter stationer. De har även haft en mindre varians vilket går att tolka på ett sådant sätt att anropen är mer förutsägbara ur ett tidsperspektiv.

4.3 Val av actor-modell

Akka.NET har en strikt hierarki som bör följas när Akka.NET actorer används. Att använda Service Fabric Reliable Actors bedömdes delvis därför vara mer anpassat för att bygga på lönemotorn. En stark anledning till valet av modell var att lönemotorn redan var baserad på Microsoft Azure och att den därför var mindre komplicerad att anpassa till Microsoft Azure Service Fabric Reliable Actors.

Genom att Service Fabric Reliable Actors valdes kunde actorer ersatta delar av det befintliga programmet. Med Akka.NETs rekommenderade strikta hierarki bedömdes det vara svårare att använda dem som enskilda inslag i det befintliga programmet då

Akka.NETs actorer skulle kräva att programmet byggdes om från början för att vara uppbyggd på actorer.

5. Diskussion

5.1 Förståelse av kod

Huruvida koden för hämtning av *NewsActor* har förenklats beror definitionen av förenklats. Det som syftats till i detta projekt är en tydlighet i vad som gör vad och att modellen ska gå enklare att förstå. Koden innehöll efter introduceringen av Service Fabric Reliable Actorn *NewsActor* mer kod och fler olika filer med tillägget av *NewsActor* och *INewsActor*. Det som ledde till den längre koden var till stor del formalia kring *NewsActor* och dess interface *INewsActor.* Bland annat innehöll interfacet en klass för att identifiera actorn och i actorn fanns formella funktioner som *OnActivateAsync()* samt *OnDeactivateAsync()*.

Vid en första anblick kan detta försvåra förståelsen av koden men vid ett scenario där actor-modellen är implementerad är detta en struktur som återkommer för alla actorer. Det är alltså något som i längden kan bidra till igenkännande när koden studeras. En annan anledning till längden på koden var att ett val gjordes att låta nyheterna sparas i *NewsActor*. Den lagringen och tillhörande timern har endast en tidsbesparande funktion och kan därför bortses från vid studerandet av hur koden förändrats.

Att koden delades upp i fler filer kan anses ha lett till att varje del blivit mer entydig. De extra filerna *NewsActor* och *INewsActor* kan dessutom antas vara välbekanta efter en implementation av actor-modeller Service Fabric Reliable Actor eftersom alla actorer i ett sådant läge vore uppbyggda på samma sätt. *NewsService* var redan från början en enkel del av koden, med endast en funktion. Vid implementationer i mer komplicerade delar av koden bör samma typer av vinster kunna göras.

5.2 Tidspåverkan

I första anropet med hjälp av en Service Fabric Reliable Actor är det väntat att anropet tar en längre tid eftersom actorn både skapas och actorn i sin tur anropar databasen efter informationen som efterfrågas. I nästkommande anrop har actorn lagrat informationen och därmed bör nästkommande anrop att ta kortare tid jämfört med om actorn åter skulle ha behövt hämta information från databasen igen. Även för anropen utan en Service Fabric Reliable Actor är det väntat att anropen efter första anropet tar kortare tid än första anropet eftersom en form av cache används inom lönemotorn.

5.2.1 Påverkan från WorkingShiftsActor

Resultatet av testningen av anrop efter arbetspass med *WorkingShiftsActor*, se tabell 1, tog nästan 7,8 s vilket alltså var 1,3 s längre än utan actor och 7 632 ms längre tid än de resterande anropen efter arbetspass med en actor. Att första anropet tog så lång tid beror troligen på att *WorkingShiftsActor*'n inte bara kontaktade databasen för att fråga efter information utan även aktiverades. I aktiveringen av actorn skapas en datakatalog som sedan används för att hämta information från databasen. Den sparas direkt i en lista med

arbetspassobjekt i actorn, listan är en instansvariabel i *WorkingShiftsActor*. De följande anropen till *WorkingShiftsActor*'n ger listan med objekt som returvärde vilket är en operation som bör ta kortare tid än skapandet av en datakatalog och anrop till databasen. Att de följande anropen är mycket snabbare än det första anropet är alltså väntat.

De följande anropen var 0,8 ms snabbare än motsvarande utan en Service Fabric Reliable Actor. Inte heller de följande anropen utan actor hämtar direkt från databasen utan använder ett cacheminne vilket gör att även de anropen går snabbare än det första anropet utan actor. Anropen med en Service Fabric Reliable Actor gick, sett till väntevärdet för de genomsnittliga tiderna i testerna, på 4% snabbare tid än anropen utan. Men med tanke på variansen de genomsnittliga tiderna utan användandet av *WorkingShiftsActor* hade på 0,7 ms^2 är skillnaden i väntevärde på 0,8 ms inte en stor skillnad. Däremot var variansen för de genomsnittliga tiderna för anropen med en *WorkingShiftsActor* endast 0,06 ms^2. Den genomsnittliga tiden för ett anrop när man använder en actor verkar alltså vara mer konsekvent med hjälp av en Service Fabric Reliable Actor i dessa tester.

De 1680 anrop som krävs för att ta igen den längre tiden första anropet pekar på att användandet av en *WorkingShiftsActor* inte skulle spara tid som den ser ut nu. För att kunna använda den som en tidsbesparande åtgärd måste första anropet kunna förkortas. Däremot visar den lägre variansen både för första anropet och för den genomsnittliga tiden för resterande anrop på att de olika observationerna ligger närmare medelvärdet för observationerna för testerna med actor än för testerna utan actor. Det går att tolka de testesterna som att de varierar mindre med hjälp av en actor och att användandet av actor gjorde dessa anrop mer konsekventa sett till tidsåtgång.

5.2.2 Påverkan från EmployeesActor

I testen av anrop efter anställda visade enligt tabell 2 på att första anropen tog inte bara mycket längre tid än de följande anropen utan även längre tid med än första anropet utan en Service Fabric Reliable Actor. Detta kan antas bero delvis på att aktiverandet av Service Fabric Reliable Actor tar längre tid än hämtningen av informationen från databasen. Under aktiveringen av *EmployeesActor*'n skapas en datakatalog genom vilken alla anställda som hörs till företaget hämtas till en lista med objekt som lagras i actorn. Det var alltså väntat att detta första anrop tog relativt lång tid. I de följande anropen är returvärdet från *EmployeesActor*'n endast de objekten i den lagrade listan med rätt identifieringsnummer. Testen av anrop efter anställda visade också på att den genomsnittliga tiden för de följande anropen genom *EmployeesActor*'n hade ett väntevärde på 20,0 ms och även hade en väldigt låg varians av den genomsnittliga tiden sett till de fyra testerna. Det gick alltså snabbare i jämförelse med testerna utan en actor och den genomsnittliga tiden var mindre varierande. Att den genomsnittliga tiden var mindre varierande mellan testerna kan bero på att inget anrop till någon databas eller cacheminne behövde göras, det var alltså mindre faktorer som kunde påverka hur lång tid anropen tog än för anrop som gjordes utan en Service Fabric Reliable Actor. Variationsbredden för testerna där *EmployeesActor*'n användes var mindre än hälften så

stor som för testerna utan actor. Även det antyder att hur lång tid anropet tog var mer konsekvent än för testerna för motsvarande anrop utan *EmployeesActor*'n.

Enligt testerna av anrop efter anställda skulle det krävas att över 1600 anrop skulle behöva göras för att det skulle löna sig tidsmässigt att använda sig av *EmployeesActor*'n. För att *EmployeesActor*'n ska vara användbar behöver alltså tiden som aktiverandet av actorn och första anropet tar förkortas på något sätt.

5.2.3 Påverkan från LockActor

I aktiveringen av *LockActor* så skapas en datakatalog som används för att hämta information om låsta datum och spara dem i en lista med *PayrollRunLockedDate* -objekt. Listan är en instansvariabel i *LockActor*-klassen och initieras i samband med aktiveringen av actor-instansen. I de kommande anropen ger *LockActor'*n ett returvärde som är den sparade listan med objekt. Därmed är det även för *LockActor'*n väntat att första anropet kommer ta längre tid än de följande anropen. Det resultatet framkom även i testerna av anropen, se tabell 3, efter låsta datum där första anropet till *LockActor'*n efter låsta datum hade ett väntevärde på 2,5 s medan motsvarande för den genomsnittliga tiden för de kommande anropen var 13,9 ms. Testerna av anrop efter låsta datum visade alltså på att *LockActor'*n kan spara tid både på första anropet och de på det första anropet följande anropen.

5.2.4 Påverkan från StationsActor

Testerna av anrop efter stationer utmärkte sig jämfört med övriga mätningar, se tabell 4. De visade på kortare tider för anrop utan actor än med hjälp av *StationsActor* inte bara för första anropet utan även i övriga. Skillnaden i första anropet var markant, det tog omkring 63 gånger längre tid i testerna med en *StationsActor.* Jämfört med testerna av de andra anropen var det första anropet utan actor som var anmärkningsvärt snabbt med ett väntevärde på 71,2 ms. De uppmätta tiderna för första anropet till en *StationsActor* är jämförbara med de uppmätta tiderna för de första anropen i de andra testerna.

I aktiveringen av *StationsActor* skapas i likhet med de andra testade actorerna en datakatalog som i aktiveringen används för att hämta alla stationer och lagra dem i en lista med Station-objekt. De kommande anropen till *StationsActor* fungerar på samma sätt som för de andra testade actorerna att returvärdet hämtas från listan som sparats som en instansvariabel.

Den genomsnittliga tiden för de på det första anropet följande anropen var även de kortare utan hjälp från en Service Fabric Reliable Actor. Även där var det anropen utan actor som stack ut i jämförelse med de andra testerna som ovanligt korta. Den genomsnittliga tiden med actor var något kortare men jämförbart med övriga tester och de hade även jämförbar varians. Anropen efter stationer utan hjälp från *StationsActor* var däremot de snabbaste av alla tester och hade en väldigt låg varians.

Det går att tolka resultatet för testerna av anrop efter stationer som att det anropet utan hjälp från en Service Fabric Reliable Actor är effektivt och så pass snabbt att det användandet av en actor lägger till mer tid än vad det kan öka effektiviteten med.

5.3 Metodval

Att först göra simuleringar i två olika typer av actor-modeller gav dels kännedom inför simuleringen i hur de olika typerna av actor-modeller fungerade, dels kunskap om hur en actor kan byggas upp. En nackdel med det metodvalet var dock att det tog lång tid både att införskaffa kunskap om två modeller och att göra simuleringar med bägge modeller.

5.3.1 Implementation av proof-of-concept

Implementationen av actorer i lönemotorn kom att till stor del anpassas till hur den befintliga koden såg ut. Det ledde till att avsteg gjordes från actor-modellen för att kunna anpassa utformningen efter den aktuella situationen. Ett annat angreppssätt för att kunna följa actor-modellen istället för att anpassa actorer till den befintliga koden vore att strukturera om den befintliga koden från början. Det vore dock ett omfattande arbete eftersom lönemotorn innefattar flertalet projekt och extremt stora mängder kod.

Det implementerade proof-of-concept byggde endast på get-funktioner. I en potentiell vidareutveckling till set-funktioner skulle ändringar som gjordes av informationen i actorn även ändra i databasen. Därigenom skulle informationen i databasen hela tiden vara adekvat och uppdaterad. Men i denna implementation har endast get-funktioner i de olika actorerna skapats och testats inom projektet. De slutsatser som har dragits bygger alltså endast på actorernas get-funktioner.

Ett exempel på en avvägning som gjordes var att använda sig av en actor som hämtade alla anställda från databasen. Alternativet som övervägdes i början av projektet var att låta en actor hämta en anställd och alltså använda flera instanser av actorn för att hämta upp alla anställda. Det förstnämnda valdes för att den funktion som oftast anropades i lönemotorn var en som frågade efter en lista med alla anställda. Resultatet visade på att det första anropet efter anställda tog väldigt lång tid i förhållande till följande anrop. En anledning till detta skulle kunna vara att det var ett stort anrop till databasen. Därmed skulle ett sätt att förkorta tiden för det första anropet vara att låta flera instanser av en actor som endast hämtade en anställd göra varsitt mindre anrop till databasen.

5.3.2 Modellval

Den strikta strukturen för hur ett actor-system bör byggas upp i Akka.NET var en av de avgörande faktorerna för varför valet föll på Service Fabric Reliable Actor. Med den metoden kunde implementationen enklare anpassas efter hur lönemotorn var uppbyggd. Därmed gick det även att implementera enstaka actorer i lönemotorn som ett proof-of-concept utan att strukturera om hela lönemotorn. Att strukturera om hela koden för att

anpassas helt till actor-modellen bedömdes vara ett för omfattande arbete och utanför projektets syfte.

En annan avgörande faktor för valet av Service Fabric Reliable Actor var att lönemotorn redan var baserad i Microsoft Azure. Att använda sig av Microsoft Azure's actor-modell Service Fabric Reliable Actor ansågs därför kunna underlätta implementationen av actorer i lönemotorn. Detta framförallt när enstaka actorer skulle kunna användas i den befintliga koden.

Skillnaden i hur actor-modellerna lagrar sitt state är en relativt poängterad skillnad i genomgången av de olika modellerna. I implementationen av ett proof-of-concept kom det dock inte att spela någon roll när nivån av state-lagring som valdes var ”Ej Beständigt state” och all information lagrades som vanliga instansvariabler som ej var skyddade ifall actorn skulle gå ner. Detta val gjordes eftersom det ej fanns ett behov av säker lagring när informationen enkelt kunde hämtas upp ur databasen igen.

5.4 Slutdiskussion

5.4.1 Tidsmätningar

Ett återkommande resultat i testerna var att variansen för både första anropet och genomsnittet för följande anrop var mindre när en Service Fabric Reliable Actor användes. En tolkning av det resultatet ger att det går att få en mer förutsägbar längd på anropen med hjälp av en Service Fabric Reliable Actor.

I flertalet av testerna återkom resultatet att den genomsnittliga tiden för anropen efter första anropet blev lägre med hjälp av Service Fabric Reliable Actor. Men ett problem för flera av de testade actorerna var att första anropet tog så pass mycket längre tid att det skulle kräva väldigt många anrop (1600–1700) för att den kortare tiden för övriga anrop skulle kunna spara in den tiden.

Även testerna av anrop utan actorer visade på att första anropet tog mycket längre tid än genomsnittet för följande anrop. Nämligen 8 gånger längre tid för anrop efter stationer och 140–320 gånger längre tid för övriga anrop. Därmed går det att överväga att ett effektivare sätt att minska tidsåtgången för anrop och öka prestandan vore att fokusera på att korta tiden för första anropet.

Testerna av de olika anropen visade på att Service Fabric Reliable Actor fungerar olika bra till olika anrop. I stort var de olika anropen ganska lika uppbyggda och de olika actorerna byggda på liknande vis som varandra. Men ändock ledde det till olika resultat, som alltså kan anses bero på olikheter i anropen. Det anrop Service Fabric Reliable Actor fungerade sämst för, av de testade anropen, var anropet efter stationer. Att det anropet försämrades tidsmässigt av att en actor användes berodde till största del på att det första anropet tog längre tid. För att actorerna ska kunna anses spara tid och öka prestanda måste första anropet kunna tjänas in. För de actorer där första anropet hade

tagit 1600–1700 anrop att ta igen är detta alltså inte en lämplig åtgärd om ej första anropet kan kortas.

5.4.2 Utveckling

Fokuset för testerna i denna undersökning har varit hur varje anrop påverkats tidsmässigt av användande av en Service Fabric Reliable Actor. Mindre fokus har lagts på andra fördelar som en actor-modell kan leda till. Fördelar som exempelvis rör hur samtida anrop kan hanteras när flera instanser av samma actor kan verka samtidigt. Andra fördelar handlar om hur ett system på actor-modellen kan gynnas när det är dags för systemet att skala upp.

5.4.3 Avslutning

För att kunna använda actor-modellen fullt ut, även actor-modellen i mindre strikt mening, tyder denna undersökning på att det är viktigt att problemet är anpassat till actor-modellen. Alternativt att lösningen byggs från grunden för att kunna följa actor-modellen istället för att modellen ska anpassas mycket efter den befintliga situationen.

Ett genomgående resultat i testerna visade på att anropen med Service Fabric Reliable Actors hade en lägre varians än motsvarande anrop utan Service Fabric Reliable Actors. Mindre varians gör det enklare att räkna på hur lång tid något bör ta vilket gör att detta skulle kunna vara en fördel med användandet av Service Fabric Reliable Actors.

En faktor som inte tagits med i diskussionen av testerna är att den befintliga koden och cache-minnet har utvecklats och finslipats i åratal till skillnad från implementationen av Service Fabric Reliable Actors. Det kan tydas som att anropen med den befintliga koden har haft fördelen av att vara bearbetad flertalet gånger innan och att koden med Service Fabric Reliable Actors skulle kunna gynnas med mer genomarbetning. En längre tids bearbetning skulle eventuellt kunna leda till att både de första, ganska långa, anropen och de följande, redan relativt korta, anropen förbättrades.

6. Slutsatser

6.1 Metodval

Att implementera actor-modellen med Service Fabric Reliable Actor i ett befintligt program innebar att actor-modellen anpassades efter tidigare kod. Det gjordes alltså anpassningar av modellen för att passa problemet. För att helt kunna använda sig av en actor-modell, även en mindre strikt version av en actor-modell, kan större delar av koden behöva anpassas efter modellen. Alternativt kan actor-modellen gynnas av att koden från början skrivs eller skrivs om helt efter actor-modellen.

Valet att använda Service Fabric Reliable Actor var till största del underbyggt av hur lönemotorn redan var uppbyggd. Den hostades redan av Microsoft Azure vilket bidrog till att det bedömdes vara en lämplig väg att använda Service Fabric Reliable Actor. En annan faktor var att det för Akka.NET framkom att det rekommenderades en relativt strikt struktur. Därmed bedömdes Service Fabric Reliable Actor vara ett lämpligare verktyg för att omvandla delar av programmet till actor-modell och utföra tester. Som verktyg att skapa ett effektivt system uppbyggd på actorer har inte modellerna vägts mot varandra i denna undersökning, det har varit situationen som avgjort modellvalet. Service Fabric Reliable Actor istället för Akka.NET blev därför ganska givet sett till anpassningarna som behövde göras för att passa det befintliga problemet. Med Akka.NET, som förespråkar en strikt struktur, tyder det på att anpassningarna efter befintlig kod hade varit försvårade.

Simuleringarna innan gav en bild av de olika modellerna som endast teori ej hade gett. Det gav även ett perspektiv av actor-modellen som visade på skillnaden att i implementationen användes actorerna mer anpassat efter den befintliga koden.

6.2 Resultat

Genom att använda sig av actorer och actor-modellen blir det väldigt tydligt vad varje del av koden gör eftersom varje actor i en strikt actor-modell endast ska göra en sak. Även i en mindre strikt actor-modell är det viktigt att en actor håller sig till att göra en begränsad uppgift. Genom att actor-modellen i den undersökta applikationen implementerades i en redan existerande lösning var det inte möjligt att hålla en strikt actor-modell. Actor-modellen kan ändock anses ha gjort koden mer översiktsbar.

I de försök som gjorts i denna undersökning kan man dra slutsatsen att det krävs många upprepande anrop för att en actor-modell ska bidra till en snabbare respons i den undersökta situationen. För att de aktuella actorerna ska kunna ge en snabbare respons inom ett mer rimligt antal upprepade anrop skulle det krävas att tiden för första anropet förkortas. Däremot visade testerna att variansen kan minska och en mer konsekvent anropstid kan uppnås

Referenser

Akka.NET (2017a) “Get Akka.NET” https://getakka.net/articles/concepts/actor-systems.html (Hämtad 2018-06-16)

Akka.NET (2017b) “Get Akka.NET” https://getakka.net/articles/intro/tutorial-1.html (Hämtad 2018-06-16)

Akka.NET (2017c) “Get Akka.NET” https://getakka.net/articles/actors/untyped-actor-api.html#actor-lifecycle (Hämtad 2018-06-20)

Akka.NET (2017d) “Get Akka.NET” https://doc.akka.io/docs/akka/2.5.3/scala/guide/actors-intro.html (Hämtad 2018-07-20)

Bai, H. (2016). "Programming Microsoft Azure Service Fabric." USA: Microsoft Press.

Cordemans, P., Steegmans, E. & Boydens, J. (2015) "Deterministically testing actor-based concurrent software", A-TEST 2015 Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation, pp. 26-30, ACM

De Koster, J., Marr, S., Van Cutsem, T., & D'Hondt, T. (2015). ”Domains: Sharing state in the communicating event-loop.” Computer languages, systems & structures. Volume 45, pp 132-160, April 2016. https://doi.org/10.1016/j.cl.2016.01.003

Garnock-Jones, T. (den 19 10 2016). "History of Actors." Harvard. https://www.seas.harvard.edu/courses/cs252/2016fa/12.pdf (Hämtad 2018-07-23)

Microsoft Dokumentation (2017a) “Using Data Contracts” https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/using-data-contracts (Hämtad 2018-06-18)

Microsoft Dokumentation (2017b) “Introduction to Service Fabric Reliable Actors” https://docs.microsoft.com/sv-se/azure/service-fabric/service-fabric-reliable-actors-introduction (Hämtad 2018-06-18)

Microsoft Dokumentation (2017c) “Stopwatch Class” https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch(v=vs.110).aspx (Hämtad 2017-07-24)

Sørensen, Claus Asbjørn (2015) “Azure Service Fabric introduction - Part 4: Actors” https://blog.geist.no/azure-service-fabric-introduction-part-4-actors/ (Hämtad 2018-06-18)

Turecek, V. & Delgado, L. (2016), “Reliable Actors state management”, GitHub article, https://github.com/uglide/azure-content/blob/master/articles/service-fabric/service-fabric-reliable-actors-state-management.md (Hämtad 2018-06-11)

Appendix A – Kod för tidsmätningar

```
namespace TimeTest
{
    /// <summary>
    /// An instance of this class is created for each service instance by the Service Fabric
runtime.
    /// </summary>
    internal sealed class TimeTest : StatelessService
    {
        public List<String> toWrite;
        IEmployeesActor _empActor;
        IEnumerable<long> ids;
        IEmployeeAsyncService _employeeService;
        IWorkingShiftAsyncService _workingShiftService;
        INewsService _newsService;
        IPayrollRunLockedDateStoreService _payrollRunLockedDateStoreService;
        IStationAsyncService _stationAsyncService;
        string _systemName="_konto";
        string[] cats;
        //---------- For EmpService
        private readonly IAsyncStore<Employee> store;
        private IAsyncStore<WorkingShift> storeWS;
        private IAsyncStore<PayrollRunLockedDate> PLDstore;
        private StationStore Sstore;
        private IAuthProvider authProvider;
        ITrackedEntityService trackedEntityService;
        IWorkingShiftAsyncService workingShiftService;
        IEmployeeRepository employeeRepository;
        ISystemPayrollInformationService systemPayrollInformationService;
        IPayrollRunLockedDateStoreService payrollRunLockedDateStoreService;
        ISettingsRepository settingsRepository;
        IEmployeeEnableService employeeEnableService;
        IPayrollArticleInstanceStatisticsService payrollArticleInstanceStatisticsService;
        IExtraProfessionStatisticsService extraProfessionStatisticsService;
        ISystemNameProvider systemNameProvider;
        IWhitelistService whiteListService;
        IUserService userService;
        IRoleTreeAsyncService roleTreeService;
        IImageService imageService;
        IPayrollRunCache payrollRunCache;
        IMarcExceptionLogger exceptionLogger;
        ICompanyInformationRepository companyInformationRepository;
        IInternalEmploymentService internalEmploymentService;
        IRegionFunctionsManager regionFunctionsManager;
        //--------- For WS Service
        IWorkingShiftRepository workingShiftRepository;
        IWorkingShiftHistoryAsyncService workingShiftHistoryService;
```

```
        IWorkingShiftHistoryRepository workingShiftHistoryRepository;
        ICacheHelperFactory _cacheHelperFactory;


        // --------------
        public TimeTest(StatelessServiceContext context)
            : base(context)
        { }

        /// <summary>
        /// Optional override to create listeners (e.g., TCP, HTTP) for this service replica to
handle client or user requests.
        /// </summary>
        /// <returns>A collection of listeners.</returns>
        protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
        {
            return new ServiceInstanceListener[0];
        }

        /// <summary>
        /// This is the main entry point for your service instance.
        /// </summary>
        /// <param name="cancellationToken">Canceled when Service Fabric needs to shut down this
service instance.</param>
        ///
        protected override async Task RunAsync(CancellationToken cancellationToken)
        {
            ids = new List<long> { 1,2,3,4,5,6 };
            toWrite = new List<string>();

            var mockAccessProvider = new FullAccessProvider(system: "_konto");
            authProvider = new FullAccessProvider(identity: new AnonymousTestIdentity(), system:
_systemName);
            regionFunctionsManager = new RegionFunctionsManager(mockAccessProvider);
            _employeeService = new
EmployeeService(store,authProvider,trackedEntityService,workingShiftService,employeeRepository,s
ystemPayrollInformationService,payrollRunLockedDateStoreService,settingsRepository,employeeEnabl
eService,payrollArticleInstanceStatisticsService,extraProfessionStatisticsService,systemNameProv
ider,whiteListService,userService,roleTreeService,imageService,payrollRunCache,exceptionLogger,c
ompanyInformationRepository,internalEmploymentService,regionFunctionsManager);

            _cacheHelperFactory = new NullCacheHelperFactory();

            storeWS = new Store<WorkingShift>(new MockWorkingShiftRepository(), new
HistoryService<WorkingShift>(null,null,null), null, null, null, _cacheHelperFactory, null,
null);

            _workingShiftService = new WorkingShiftService(storeWS, authProvider,
workingShiftRepository, workingShiftHistoryService, workingShiftHistoryRepository,
```

```
payrollRunLockedDateStoreService, employeeEnableService, systemPayrollInformationService,
payrollRunCache);

            _newsService = new NewsService();
            cats = new string[1] { "mobile - salery"};

            _payrollRunLockedDateStoreService = new PayrollRunLockedDateStoreService(PLDstore,
authProvider, null, null, null);


            IStationRepository _sRepo = new MockStationRepository();
            var mockPermissionSetProvider = new MockPermissionSetProvider();
            var mockAuditLogService = new MockAuditLogService();
            var mockCache = new MockCache();
            var mockCacheHelperFactory = new NullCacheHelperFactory();
            var mockEventManager = new MockEventManager();
            var mockTransaction = new MockTransaction();
            var stationHistoryService = new HistoryService<Station>(new
MockHistoryRepository<Station>(), _sRepo, mockPermissionSetProvider);
            Sstore = new StationStore(_sRepo, stationHistoryService, mockAuditLogService,
mockPermissionSetProvider, mockCache, mockCacheHelperFactory, mockEventManager,
mockTransaction);
            _stationAsyncService = new StationService(Sstore, authProvider, null, null, null);

            TimeOperations();




        }
        internal class AnonymousTestIdentity : Identity
        {
            public AnonymousTestIdentity()
            {
                this.Name = "Anon test identity";
                this.Subject = Guid.Empty;
                this.Type = IdentityType.Anonymous;
            }
        }

        //---------------------------For OpTimer:
        private async void TimeOperations()
        {
            long nanosecPerTick = (1000L * 1000L * 1000L) / Stopwatch.Frequency;
            const long numIterations = 10000;

            // Define variables for operation statistics.
            long numTicks = 0;
            long numRollovers = 0;
```

```
            long maxTicks = 0;
            long minTicks = Int64.MaxValue;
            int indexFastest = -1;
            int indexSlowest = -1;
            int indexNextSlowest = -1;
            long milliSec = 0;
            long firstTicks = 0;
            long secondMaxTicks = 0;

            Stopwatch time10kOperations = Stopwatch.StartNew();

            for (int i = 0; i <= numIterations - 1; i++)
            {
                long ticksThisTime = 0;
                Stopwatch timePerParse;
                // Start a new stopwatch timer.
                timePerParse = Stopwatch.StartNew();

                //-----------------Code to meassure
                Thread.Sleep(10);

                //---WS, SF/not - in service
                //await _workingShiftService.GetAsync(new QueryOptions(),
TestHelpers.GetStandardSwedishSystemPayrollInformation(true, true));
                //----------

                //-- NewsService, SF/not in service
                //await _newsService.GetAsync(cats, "sv");
                //----

                //-- NewsService, SF/not in service
                //await _stationAsyncService.GetAsync(new QueryOptions());
                //----

                //-- LockDate, SF/not in service
                await _payrollRunLockedDateStoreService.GetPayrollRunLockedDates();
                //----


                // ---------- Get Emp
                //await _employeeService.GetAsync(ids);
                // -------------- End Emp

                //-------------------------------- End of code
                // Stop the timer, and save the
                // elapsed ticks for the operation.

                timePerParse.Stop();
                ticksThisTime = timePerParse.ElapsedTicks;
```

```
                // Update operation statistics
                // for iterations 1-10001.
                if (i == 0)
                {
                    firstTicks = ticksThisTime;
                }

                if (secondMaxTicks<ticksThisTime && ticksThisTime<=maxTicks && maxTicks != -1)
                {
                    indexNextSlowest = i;
                    secondMaxTicks = ticksThisTime;
                }

                if (maxTicks < ticksThisTime)
                {
                    indexSlowest = i;
                    maxTicks = ticksThisTime;
                }
                if (minTicks > ticksThisTime)
                {
                    indexFastest = i;
                    minTicks = ticksThisTime;
                }
                numTicks += ticksThisTime;
                if (numTicks < ticksThisTime)
                {
                    // Keep track of rollovers.
                    numRollovers++;
                }
            }

            time10kOperations.Stop();
            milliSec = time10kOperations.ElapsedMilliseconds;

            // Write statistics to document

            toWrite.Add("Summary:");
            toWrite.Add(String.Format("First time: {0} ticks = {1} nanoseconds", firstTicks,
firstTicks * nanosecPerTick));
            toWrite.Add(String.Format("  Slowest time:  #{0}/{1} = {2} ticks= {3} nanoseconds",
            indexSlowest, numIterations, maxTicks, maxTicks* nanosecPerTick));
            toWrite.Add(String.Format("  Second slowest time:  #{0}/{1} = {2} ticks= {3}
nanoseconds",
                indexNextSlowest, numIterations, secondMaxTicks, secondMaxTicks *
nanosecPerTick));
            toWrite.Add(String.Format("  Fastest time:  #{0}/{1} = {2} ticks= {3} nanoseconds",
```

```
                indexFastest, numIterations, minTicks, minTicks* nanosecPerTick));
            toWrite.Add(String.Format("  Average time:  {0} ticks = {1} nanoseconds",
                numTicks / numIterations,
                (numTicks * nanosecPerTick) / numIterations));
            toWrite.Add(String.Format("  Average time except first:  {0} ticks = {1}
nanoseconds",
                (numTicks - firstTicks) / numIterations,
                ((numTicks - firstTicks) * nanosecPerTick) / (numIterations - 1)));
            toWrite.Add(String.Format("  Total time looping through {0} operations: {1}
milliseconds",
                numIterations, milliSec));


            using (StreamWriter outputFile = new
StreamWriter(@"\\caspeco.se\upp\home\sofie.mahler\Documents\WriteTo\latest.txt"))
            {
                foreach (string line in toWrite)
                {
                    outputFile.WriteLine(line);
                }
            }

        }
        private void DisplayTimerProperties()
        {
            // Display the timer frequency and resolution.
            if (Stopwatch.IsHighResolution)
            {
                toWrite.Add("Operations timed using the system's high-resolution performance
counter.");
            }
            else
            {
                toWrite.Add("Operations timed using the DateTime class.");
            }

            long frequency = Stopwatch.Frequency;
            toWrite.Add(String.Format("  Timer frequency in ticks per second = {0}",
                frequency));
            long nanosecPerTick = (1000L * 1000L * 1000L) / frequency;
            toWrite.Add(String.Format("  Timer is accurate within {0} nanoseconds",
                nanosecPerTick));
        }


        //--------------------------------- End OpTimer
    }
}
```

Appendix B – Kod för simulering med Akka.NET actorer

```
namespace ActorTest
{
    class Program
    {
        public static ActorSystem MySystem;


        static void Main(string[] args)
        {
            MySystem = ActorSystem.Create("MySystem");
            int custOne = 1;


            IActorRef reportGenActor = MySystem.ActorOf(Props.Create(() => new
ReportGenActor(custOne)), "reportGenActor");

            reportGenActor.Tell(new Messages(ReportGenActor.StartCommand));

            Console.ReadLine();
        }
    }

}

namespace ActorTest
{
    class Messages
    {
        public Messages(string mess)
        {
            Mess = mess;
        }
        public string Mess { get; private set; }


        public class InfoMess
        {
            public InfoMess(List<int> emp, Dictionary<int, int> ws)
            {
                Employees = emp;
                WorkShifts = ws;
            }
            public IReadOnlyList<int> Employees { get; private set; }
            public IReadOnlyDictionary<int, int> WorkShifts { get; private set; }

        }
        public class GetInfoMess
        {
            public GetInfoMess()
            {
            }
        }
        public class StartPayRollActor
        {
            public StartPayRollActor()
            {

            }
            public StartPayRollActor(List<ResultObject> ackResultO)
            {
                AckResultO = ackResultO;
            }
            public List<ResultObject> AckResultO { get; private set; }
        }

        public class PayRollActorAnswer
        {
```

```
            public PayRollActorAnswer(ResultObject ro)
            {
                ResultO = ro;
                AckResultO = new List<ResultObject>();
                AckResultO.Add(ro);
            }

            public PayRollActorAnswer(List<ResultObject> ackRo)
            {
                AckResultO = ackRo;
            }

            public ResultObject ResultO { get; private set; }
            public List<ResultObject> AckResultO { get; private set; }
        }

        public class ReportAgg
        {
            public ReportAgg()
            {

            }
            public ReportAgg(IActorRef actorRef)
            {
                ActorRef = actorRef;
            }
            public IActorRef ActorRef { get; private set; }
    }

        public class StartAgg
        {
            public StartAgg(List<IActorRef> allActorRefs)
            {
                AllActorRefs = allActorRefs;
            }
            public List<IActorRef> AllActorRefs { get; private set; }
        }

        public class AggAnswer
        {
            public AggAnswer()
            {
            }
            public List<IActorRef> Answers { get; private set; }

        }
    }
}

namespace ActorTest
{
    class AggActor : UntypedActor
    {
        List<IActorRef> AllActorRefs = new List<IActorRef>();
        List<IActorRef> DoneActorRefs = new List<IActorRef>();
        IActorRef requester;


        protected override void OnReceive(object message)
        {
            if (message is Messages.StartAgg)
            {
                Messages.StartAgg msg = message as Messages.StartAgg;
                AllActorRefs = msg.AllActorRefs;
                requester = Sender;
            }

            else if (message is Messages.ReportAgg)
            {
                Messages.ReportAgg msg = message as Messages.ReportAgg;
                if (msg.ActorRef.IsNobody())
                {
                    DoneActorRefs.Add(Sender);
                    Console.WriteLine(Sender.ToString() + "is done, from AggActor");
```

```
                }
                else
                {
                    DoneActorRefs.Add(msg.ActorRef);
                    Console.WriteLine(msg.ActorRef.ToString() + "is done, from AggActor");
                }


                CheckIfDone();

            }
        }
        private void CheckIfDone()
        {
            if ((DoneActorRefs != null) && (AllActorRefs != null))
            {
                var areEquivalent =
!AllActorRefs.Except(DoneActorRefs).Union(DoneActorRefs.Except(AllActorRefs)).Any();
                if (areEquivalent)
                {
                    requester.Tell(new Messages.AggAnswer());
                }
            }

        }

    }
}

namespace ActorTest
{
    public class PayRollActor : UntypedActor
    {
        private readonly IActorRef _payRollActorCoordinator;

        public int Customer { get; private set; }
        public int EmpID { get; private set; }
        public int LastMonth { get; private set; }

        List<ResultObject> ackResult;


        public PayRollActor(IActorRef payRollActorCoordinator, int customer, int empId, int
lastMonth)
        {
            _payRollActorCoordinator = payRollActorCoordinator;
            Customer = customer;
            EmpID = empId;
            LastMonth = lastMonth;
        }

        protected override void OnReceive(object message)
        {
            if (message is Messages.StartPayRollActor msg)
            {
                Calculate(EmpID);
            }

            else if (message is Messages.PayRollActorAnswer answerMsg)
            {
                ackResult = answerMsg.AckResultO;
                var lTemp = new List<int>();
                foreach (ResultObject ro in ackResult)
                {
                    lTemp.Add(ro.temp);
                }
                Console.Write("From " + Self.Path + ":");
                lTemp.ForEach(Console.Write);
                Console.WriteLine();
                _payRollActorCoordinator.Tell(answerMsg, Self);

            }

        }
```

```
        private void Calculate(int id)
        {

            System.Threading.Thread.Sleep(1000);

            string name = String.Format("PayRollMonthActor{0}",LastMonth);
            IActorRef payRollMonthActor = CreateIfNotExist(Context, name);
            payRollMonthActor.Tell(new Messages.StartPayRollActor(), Self);

        }
        private IActorRef CreateIfNotExist(IActorContext context, string name)
        {
            var actor = context.Child(name);
            if (actor.Equals(ActorRefs.Nobody))
            {
                Console.WriteLine("New actor " + name);
                return context.ActorOf(Props.Create(() => new PayRollMonthActor(Self, Customer,
EmpID, LastMonth)), name);
            }
            Console.WriteLine("Old actor " + name);
            return actor;
        }

    }

}

namespace ActorTest
{
    class PayRollActorCoordinator:UntypedActor
    {
        IActorRef AggActor = Context.ActorOf(Props.Create(() => new AggActor()), "aggActor");
        IReadOnlyList<int> employees;
        IReadOnlyDictionary<int, int> workShifts;
        List<IActorRef> employeePayRollActors = new List<IActorRef>();
        List<ResultObject> resultObjectsFromPayRollActors = new List<ResultObject>();

        IActorRef ReportGenActor;

        public int CustomerId { get; private set; }
        public int LastMonth { get; private set; }

        public PayRollActorCoordinator(IActorRef reportGenActor, int custId, int lastMonth)
        {
            ReportGenActor = reportGenActor;
            CustomerId = custId;
            LastMonth = lastMonth;
        }
        protected override void OnReceive(object message)
        {
            if (message is Messages.InfoMess)
            {
                Messages.InfoMess msg = message as Messages.InfoMess;
                employees = msg.Employees;
                workShifts = msg.WorkShifts;
                Console.WriteLine("In RGA and got mess from IA");
                CreatePayRollActors(msg.Employees);
            }
            else if (message is Messages.PayRollActorAnswer)
            {
                Messages.PayRollActorAnswer msg = message as Messages.PayRollActorAnswer;
                resultObjectsFromPayRollActors.Add(msg.Result0);
                AggActor.Tell(new Messages.ReportAgg(Sender));
            }
            else if (message is Messages.AggAnswer)
            {
                Messages.AggAnswer msg = message as Messages.AggAnswer;
                Console.WriteLine("All done!");
            }
        }

        private void CreatePayRollActors(IReadOnlyList<int> employees)
        {
            foreach (int i in employees)
```

```
            {
                string name = String.Format("PayRollActor{0}", i);
                IActorRef payRollActor = CreateIfNotExist(Context, name, i);

                payRollActor.Tell(new Messages.StartPayRollActor());

                employeePayRollActors.Add(payRollActor);
            }
            var j = 3;
            string name2 = String.Format("PayRollActor{0}", j);
            System.Threading.Thread.Sleep(3000);
            IActorRef payRollActor2 = CreateIfNotExist(Context, name2, j);
            payRollActor2.Tell(new Messages.StartPayRollActor());

            employeePayRollActors.Add(payRollActor2);
            AggActor.Tell(new Messages.StartAgg(employeePayRollActors));


        }
        private IActorRef CreateIfNotExist(IActorContext context, string name, int j)
        {
            var actor = context.Child(name);
            if (actor.Equals(ActorRefs.Nobody))
            {
                Console.WriteLine("New actor " + name);
                return context.ActorOf(Props.Create(() => new PayRollActor(Self, CustomerId, j,
LastMonth)), name);
            }
            Console.WriteLine("Old actor " + name);
            return actor;
        }

    }
}

namespace ActorTest
{
    class PayRollMonthActor : UntypedActor
    {
        private readonly IActorRef ParentActor;

        private readonly int stopCondition = 5;

        public int Customer { get; private set; }
        public int EmpID { get; private set; }
        public int Month { get; private set; }

        private bool NextMonth { get; set; }=true;

        public ResultObject Info { get; private set; }
        public List<ResultObject> AckInfo { get; private set; }

        public PayRollMonthActor(IActorRef createActor, int custId, int empID, int month)
        {
            ParentActor = createActor;
            Customer = custId;
            EmpID = empID;
            Month = month;
            if (month <= stopCondition)
            {
                NextMonth = false;
            }

        }

        protected override void OnReceive(object message)
        {

            if (message is Messages.StartPayRollActor)
            {
                Messages.StartPayRollActor msg = message as Messages.StartPayRollActor;
                if (AckInfo != null)
                {
                    ParentActor.Tell(new Messages.PayRollActorAnswer(AckInfo), Self);
```

```
                    Console.WriteLine("Sended info collected earlier, from " + Self.Path );
                }
                else
                {
                    if (NextMonth)
                    {
                        string name = String.Format("PayRollMonthActor{0}", Month - 1);
                        IActorRef payRollMonthActor = CreateIfNotExist(Context, name);
                        payRollMonthActor.Tell(new Messages.StartPayRollActor(), Self);
                    }
                    else
                    {
                        System.Threading.Thread.Sleep(1000);
                        Info = new ResultObject(Month*EmpID);
                        ParentActor.Tell(new Messages.PayRollActorAnswer(Info), Self);
                    }
                }



            }
            else if(message is Messages.PayRollActorAnswer)
            {
                Messages.PayRollActorAnswer msg = message as Messages.PayRollActorAnswer;
                Info = new ResultObject(Month * EmpID);
                AckInfo = msg.AckResultO;
                AckInfo.Add(Info);
                ParentActor.Tell(new Messages.PayRollActorAnswer(AckInfo), Self);
            }

    }
        private IActorRef CreateIfNotExist(IActorContext context, string name)
        {
            var actor = context.Child(name);
            if (actor.Equals(ActorRefs.Nobody))
            {
                Console.WriteLine("New actor " + name);
                return context.ActorOf(Props.Create(() => new PayRollMonthActor(Self, Customer,
EmpID, Month - 1)), name);
            }
            Console.WriteLine("Old actor " + name);
            return actor;
        }

        private void Calculate()
        {

        }
    }
}

namespace ActorTest
{
    public class ReportGenActor : UntypedActor
    {
        private readonly IActorRef _infoActor;
        public const string StartCommand = "Start";
        IActorRef payRollActorCoordinator;
        public int LastMonth { get; private set; } = 10;
        public int CustomerId { get; private set; }

        public ReportGenActor(int custId)
        {
            CustomerId = custId;
            IActorRef infoActor = Context.ActorOf(Props.Create(() => new InfoActor()),
"infoActor");
            payRollActorCoordinator = Context.ActorOf(Props.Create(() => new
PayRollActorCoordinator(Self, CustomerId, LastMonth)), "payRollActorCoordinator");
            _infoActor = infoActor;

        }

        protected override void OnReceive(object message)
```

```
        {
            if(message is Messages)
            {
                Messages msg = message as Messages;
                if(msg.Mess.Equals(StartCommand))
                {
                    _infoActor.Tell(new Messages.GetInfoMess());
                    Console.WriteLine("In RGA and told IA");
                }
            }

            else if (message is Messages.InfoMess)
            {
                payRollActorCoordinator.Tell(message, Self);
            }

        }

    }
}


namespace ActorTest
{
    class ResultObject
    {
        public int temp;
        public ResultObject(int x)
        {
            temp = x;
        }


    }
}
```

Appendix C – Kod för simulering med Service Fabric Reliable Actors

C1 – namespace Actors

```
namespace Actors
{
    [EventSource(Name = "MyCompany-ActorTrySF-Actors")]
    internal sealed class ActorEventSource : EventSource
    {
        public static readonly ActorEventSource Current = new ActorEventSource();

        static ActorEventSource()
        {
            Task.Run(() => { });
        }

        // Instance constructor is private to enforce singleton semantics
        private ActorEventSource() : base() { }

        #region Keywords
.
        public static class Keywords
        {
            public const EventKeywords HostInitialization = (EventKeywords)0x1L;
        }
        #endregion

        #region Events

        [NonEvent]
        public void Message(string message, params object[] args)
        {
            if (this.IsEnabled())
            {
                string finalMessage = string.Format(message, args);
                Message(finalMessage);
            }
        }

        private const int MessageEventId = 1;
        [Event(MessageEventId, Level = EventLevel.Informational, Message = "{0}")]
        public void Message(string message)
        {
            if (this.IsEnabled())
            {
                WriteEvent(MessageEventId, message);
            }
        }

        [NonEvent]
        public void ActorMessage(Actor actor, string message, params object[] args)
        {
            if (this.IsEnabled()
                && actor.Id != null
                && actor.ActorService != null
                && actor.ActorService.Context != null
                && actor.ActorService.Context.CodePackageActivationContext != null)
            {
                string finalMessage = string.Format(message, args);
                ActorMessage(
                    actor.GetType().ToString(),
                    actor.Id.ToString(),
                    actor.ActorService.Context.CodePackageActivationContext.ApplicationTypeName,
                    actor.ActorService.Context.CodePackageActivationContext.ApplicationName,
                    actor.ActorService.Context.ServiceTypeName,
                    actor.ActorService.Context.ServiceName.ToString(),
                    actor.ActorService.Context.PartitionId,
                    actor.ActorService.Context.ReplicaId,
                    actor.ActorService.Context.NodeContext.NodeName,
```

```
                    finalMessage);
            }
        }

        // For very high-frequency events it might be advantageous to raise events using
WriteEventCore API.
        // This results in more efficient parameter handling, but requires explicit allocation
of EventData structure and unsafe code.
        // To enable this code path, define UNSAFE conditional compilation symbol and turn on
unsafe code support in project properties.
        private const int ActorMessageEventId = 2;
        [Event(ActorMessageEventId, Level = EventLevel.Informational, Message = "{9}")]
        private
#if UNSAFE
            unsafe
#endif
            void ActorMessage(
            string actorType,
            string actorId,
            string applicationTypeName,
            string applicationName,
            string serviceTypeName,
            string serviceName,
            Guid partitionId,
            long replicaOrInstanceId,
            string nodeName,
            string message)
        {
#if !UNSAFE
            WriteEvent(
                    ActorMessageEventId,
                    actorType,
                    actorId,
                    applicationTypeName,
                    applicationName,
                    serviceTypeName,
                    serviceName,
                    partitionId,
                    replicaOrInstanceId,
                    nodeName,
                    message);
#else
                const int numArgs = 10;
                fixed (char* pActorType = actorType, pActorId = actorId, pApplicationTypeName =
applicationTypeName, pApplicationName = applicationName, pServiceTypeName = serviceTypeName,
pServiceName = serviceName, pNodeName = nodeName, pMessage = message)
                {
                    EventData* eventData = stackalloc EventData[numArgs];
                    eventData[0] = new EventData { DataPointer = (IntPtr) pActorType, Size =
SizeInBytes(actorType) };
                    eventData[1] = new EventData { DataPointer = (IntPtr) pActorId, Size =
SizeInBytes(actorId) };
                    eventData[2] = new EventData { DataPointer = (IntPtr) pApplicationTypeName,
Size = SizeInBytes(applicationTypeName) };
                    eventData[3] = new EventData { DataPointer = (IntPtr) pApplicationName, Size
= SizeInBytes(applicationName) };
                    eventData[4] = new EventData { DataPointer = (IntPtr) pServiceTypeName, Size
= SizeInBytes(serviceTypeName) };
                    eventData[5] = new EventData { DataPointer = (IntPtr) pServiceName, Size =
SizeInBytes(serviceName) };
                    eventData[6] = new EventData { DataPointer = (IntPtr) (&partitionId), Size =
sizeof(Guid) };
                    eventData[7] = new EventData { DataPointer = (IntPtr)
(&replicaOrInstanceId), Size = sizeof(long) };
                    eventData[8] = new EventData { DataPointer = (IntPtr) pNodeName, Size =
SizeInBytes(nodeName) };
                    eventData[9] = new EventData { DataPointer = (IntPtr) pMessage, Size =
SizeInBytes(message) };

                    WriteEventCore(ActorMessageEventId, numArgs, eventData);
                }
#endif
        }
```

```
        private const int ActorHostInitializationFailedEventId = 3;
        [Event(ActorHostInitializationFailedEventId, Level = EventLevel.Error, Message = "Actor
host initialization failed", Keywords = Keywords.HostInitialization)]
        public void ActorHostInitializationFailed(string exception)
        {
            WriteEvent(ActorHostInitializationFailedEventId, exception);
        }
        #endregion

        #region Private Methods
#if UNSAFE
            private int SizeInBytes(string s)
            {
                if (s == null)
                {
                    return 0;
                }
                else
                {
                    return (s.Length + 1) * sizeof(char);
                }
            }
#endif
        #endregion
    }
}

namespace Actors
{
    internal static class Program
    {
        /// <summary>
        /// This is the entry point of the service host process.
        /// </summary>
        private static void Main()
        {
            try
            {
                // This line registers an Actor Service to host your actor class with the
Service Fabric runtime.
                // The contents of your ServiceManifest.xml and ApplicationManifest.xml files
                // are automatically populated when you build this project.
                // For more information, see https://aka.ms/servicefabricactorsplatform

                ActorRuntime.RegisterActorAsync<EmployeeActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<PayRollActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<PayListActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<LockActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<WorkingShiftsActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<InstancesActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<SystemSettingsActor>(
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<PayrollPreviewActor>(
```

```
                    (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                Thread.Sleep(Timeout.Infinite);
            }
            catch (Exception e)
            {
                ActorEventSource.Current.ActorHostInitializationFailed(e.ToString());
                throw;
            }
        }
    }
}

namespace Actors
{
    [StatePersistence(StatePersistence.Persisted)]
    internal class PayListActor : Actor, IPayListActor
    {
        IReadOnlyList<int> Employees;
        private string _systemName;
        private string _payDate;

        private string stateName;

        List<IPayRollActor> employeePayRollActors = new List<IPayRollActor>();

        [DataContract]
        internal sealed class PayListState
        {
            [DataMember]
            public Dictionary<int,long> PayRollResults { get; set; }
        }

        public PayListActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            var a = PayListActorId.Parse(actorId);
            _systemName = a.SystemName;
            _payDate = a.PayDate;
        }

        protected override async Task OnActivateAsync()
        {
            ActorEventSource.Current.ActorMessage(this, "Actor activated.");
            stateName = "State";
            PayListState prcState = new PayListState { PayRollResults = new Dictionary<int,
long>()};
            await StateManager.TryAddStateAsync<PayListState>(stateName, prcState);
            await StateManager.SaveStateAsync();
        }

        public async Task<Dictionary<int,long>> StartPayRollCoordinatiorAsync(CancellationToken
cancellationToken)
        {

            var state = await StateManager.GetStateAsync<PayListState>(stateName);

            if (employeePayRollActors.Any())
            {
                return state.PayRollResults;
            }
            else
            {
                var empActorId = new EmployeeActorId(_systemName);
                IEmployeeActor empActor = ActorProxy.Create<IEmployeeActor>(empActorId.Id,
serviceName: "EmployeeActorService");
                cancellationToken.ThrowIfCancellationRequested();
                Employees = await empActor.GetEmployeesAsync(cancellationToken);

                foreach (int emp in Employees)
                {
                    var actorId = new PayRollActorId(_systemName, emp, _payDate);
```

```
                    employeePayRollActors.Add(ActorProxy.Create<IPayRollActor>(actorId.Id,
serviceName: "PayRollActorService"));
                }

            }

            var tasks = new List<Task<Tuple<int, long>>>();
            foreach (IPayRollActor actor in employeePayRollActors)
            {
                tasks.Add(actor.GetResultPayRollAsync(cancellationToken));

            }
            await Task.WhenAll(tasks);

            foreach (Task<Tuple<int, long>> task in tasks)
            {
                Tuple<int, long> temp = task.Result;
                state.PayRollResults.Add(temp.Item1, temp.Item2);
            }

            await StateManager.SaveStateAsync();
            state = await StateManager.GetStateAsync<PayListState>(stateName);

            return state.PayRollResults;
        }


    }
}


namespace Actors
{
    [StatePersistence(StatePersistence.Persisted)]
    internal class EmployeeActor : Actor, IEmployeeActor
    {
        public string _systemName;


        private string stateName;

        public EmployeeActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            var a = EmployeeActorId.Parse(actorId);
            _systemName = a.SystemName;
        }

        [DataContract]
        internal sealed class EmployeesState
        {
            [DataMember]
            public List<int> Emp { get; set; }
        }

        protected override async Task OnActivateAsync()
        {
            ActorEventSource.Current.ActorMessage(this, "Actor activated.");
            await Task.Delay(500);
            stateName = "State";
            await StateManager.TryAddStateAsync<EmployeesState>(stateName, new EmployeesState {
Emp = new List<int>(new int[] { 1, 2, 3, 4, 5 }) });
            await StateManager.SaveStateAsync();
        }

        async Task<List<int>> IEmployeeActor.GetEmployeesAsync(CancellationToken
cancellationToken)
        {
            var state = await StateManager.GetStateAsync<EmployeesState>(stateName);
            return state.Emp;
        }

        async Task<List<int>> IEmployeeActor.UpdateAndGetEmployeesAsync(CancellationToken
cancellationToken)
```

```
        {
            EmployeesState empState = new EmployeesState { Emp = new List<int>(new int[] { 4, 5,
7, 8, 9 }) };
            await StateManager.AddOrUpdateStateAsync<EmployeesState>(stateName, empState, (key,
old_value) => empState, cancellationToken);
            await StateManager.SaveStateAsync(cancellationToken);
            var state = await StateManager.GetStateAsync<EmployeesState>(stateName);

            return state.Emp;
        }

    }
}


namespace Actors
{
    [StatePersistence(StatePersistence.Persisted)]
    internal class InstancesActor : Actor, IInstancesActor
    {

        [DataContract]
        internal sealed class PayListState
        {
            [DataMember]
            public Dictionary<int,long> PayRollResults { get; set; }
        }

        public InstancesActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {

        }

        protected override async Task OnActivateAsync()
        {

        }

        public async Task<int> GetInstances()
        {
            int x = 5;
            return x;
        }
    }
}

namespace Actors
{
    [StatePersistence(StatePersistence.Persisted)]
    internal class PayRollActor : Actor, IPayRollActor
    {
        private string _systemName;
        private long _employeeId;
        private string _payDate;

        private string stateName;

        [DataContract]
        internal sealed class PayRollState
        {
            [DataMember]
            public Tuple<int,long> PayRoll { get; set; }
        }

        public PayRollActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            var a = PayRollActorId.Parse(actorId);
            _systemName = a.SystemName;
            _employeeId = a.EmployeeId;
            _payDate = a.PayDate;
        }
```

```
        protected override async Task OnActivateAsync()
        {
            ActorEventSource.Current.ActorMessage(this, "Actor activated.");
            stateName = "State";
            await Task.Delay(4000);
            await StateManager.TryAddStateAsync<PayRollState>(stateName, new PayRollState {
PayRoll = new Tuple<int, long>((int)_employeeId, _employeeId * 1000) });
            await StateManager.SaveStateAsync();

        }

        async Task<Tuple<int, long>> IPayRollActor.GetResultPayRollAsync(CancellationToken
cancellationToken)
        {
            var state = await StateManager.GetStateAsync<PayRollState>(stateName);
            return state.PayRoll;
        }


    }
}


namespace Actors
{
    [StatePersistence(StatePersistence.Persisted)]
    internal class SystemSettingsActor : Actor, ISystemSettingsActor
    {

        [DataContract]
        internal sealed class PayListState
        {
            [DataMember]
            public Dictionary<int,long> PayRollResults { get; set; }
        }

        public SystemSettingsActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {

        }

        protected override async Task OnActivateAsync()
        {

        }

        public async Task<int> TestGet()
        {
            int x = 5;
            return x;
        }
    }
}
```

C2 – namespace Actors.Interfaces

```
namespace Actors.Interfaces
{

    public interface IEmployeeActor : IActor
    {
        Task <List<int>> GetEmployeesAsync(CancellationToken cancellationToken);

        Task<List<int>> UpdateAndGetEmployeesAsync(CancellationToken cancellationToken);

    }
    public class EmployeeActorId
    {
        public string SystemName;


        public EmployeeActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static EmployeeActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new EmployeeActorId(part);
        }
    }
}

namespace Actors.Interfaces
{

    public interface IInstancesActor : IActor
    {
        Task<int> GetInstances();
    }
    public class InstancesActorId
    {
        public string SystemName;

        public InstancesActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static InstancesActorId Parse(ActorId id)
```

```
        {
            var part = id.GetStringId();
            return new InstancesActorId(part);
        }
    }
}


namespace Actors.Interfaces
{

    public interface ILockActor : IActor
    {
        Task<DateTime> GetLockDate();

        Task<DateTime> SetLockDate();

        Task<int> TestGet();
    }
    public class LockActorId
    {
        public string SystemName;

        public LockActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static LockActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new LockActorId(part);
        }
    }

}


namespace Actors.Interfaces
{

    public interface IPayListActor : IActor
    {

        Task<Dictionary<int, long>> StartPayRollCoordinatiorAsync(CancellationToken
cancellationToken);
    }

        public class PayListActorId
        {
            public string SystemName;
            public string PayDate;

            public PayListActorId(string systemName, string payDate)
            {
                SystemName = systemName;
                PayDate = payDate;
            }

            public override string ToString()
```

```
            {
                return $"{SystemName};{PayDate}";
            }

            public ActorId Id
            {
                get
                {
                    return new ActorId(ToString());
                }
            }

            public static PayListActorId Parse(ActorId id)
            {
                var parts = id.GetStringId().Split(';');
                return new PayListActorId(parts[0], parts[1]);
            }
        }
    }


namespace Actors.Interfaces
{

    public interface IPayRollActor : IActor
    {
        Task<Tuple<int, long>> GetResultPayRollAsync(CancellationToken cancellationToken);

    }

    public class PayRollActorId
    {
        public string SystemName;
        public long EmployeeId;
        public string PayDate;

        public PayRollActorId(string systemName, long employeeId, string payDate)
        {
            SystemName = systemName;
            EmployeeId = employeeId;
            PayDate = payDate;
        }

        public override string ToString()
        {
            return $"{SystemName};{EmployeeId};{PayDate}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static PayRollActorId Parse(ActorId id)
        {
            var parts = id.GetStringId().Split(';');
            return new PayRollActorId(parts[0], long.Parse(parts[1]), parts[2]);
        }
    }
}

namespace Actors.Interfaces
{

    public interface ISystemSettingsActor : IActor
    {
        Task<int> TestGet();

    }
    public class SystemSettingsActorId
    {
```

```
        public string SystemName;

        public SystemSettingsActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static SystemSettingsActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new SystemSettingsActorId(part);
        }
    }
}


namespace Actors.Interfaces
{

    public interface IWorkingShiftsActor : IActor
    {
        Task<object> GetWorkingShifts(long empId, DateTime start, DateTime end);

        Task<int> TestGet();
    }
    public class WorkingShiftsActorId
    {
        public string SystemName;

        public WorkingShiftsActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static WorkingShiftsActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new WorkingShiftsActorId(part);
        }
    }
}
```

Appendix D – Namespace MarcActors från implementation i lönemotorn

D1 – Program

```
using System;
using System.Diagnostics;
using System.Fabric;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Actors.Runtime;
using Microsoft.ServiceFabric.Actors.Remoting.FabricTransport;
using Microsoft.ServiceFabric.Services.Remoting;
using Microsoft.Practices.EnterpriseLibrary.TransientFaultHandling;
using Dapper;
using System.Collections.Generic;
using Api.Models.Common.DateAndTime;
using System.Data;
using Api.Data.Dapper;

namespace MarcActors
{
    internal static class Program
    {
        /// <summary>
        /// This is the entry point of the service host process.
        /// </summary>
        private static void Main()
        {
            try
            {
                // This line registers an Actor Service to host your actor class with the
Service Fabric runtime.
                // The contents of your ServiceManifest.xml and ApplicationManifest.xml files
                // are automatically populated when you build this project.
                // For more information, see https://aka.ms/servicefabricactorsplatform

                RetryManager.SetDefault(new RetryManager(
               new List<RetryStrategy> {
                    new FixedInterval("default interactive sql", 3,
TimeSpan.FromMilliseconds(1000), true),
                    new ExponentialBackoff("default background sql", 5, TimeSpan.FromSeconds(0),
TimeSpan.FromSeconds(60), TimeSpan.FromSeconds(2), false)
               }), false);

                SqlMapper.AddTypeMap(typeof(Date), DbType.Date);
                SqlMapper.AddTypeMap(typeof(Date?), DbType.Date);
                var dateTypeHandler = new DapperDateTypeHandler();
```

```
                SqlMapper.AddTypeHandler(typeof(Date), dateTypeHandler);
                SqlMapper.AddTypeHandler(typeof(Date?), dateTypeHandler);

                SqlMapper.AddTypeHandler(DapperOffsetDateTimeHandler.Default);
                SqlMapper.AddTypeHandler(DapperLocalTimeHandler.Default);
                SqlMapper.AddTypeHandler(DapperDurationHandler.Default);


                ActorRuntime.RegisterActorAsync<DummyActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<StationsActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<NewsActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<InstancesActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<LockActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<EmployeesActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<PayrollPreviewActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<SystemSettingsActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();

                ActorRuntime.RegisterActorAsync<WorkingShiftsActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();


                ActorRuntime.RegisterActorAsync<PayrollActor>(
                   (context, actorType) => new ActorService(context,
actorType)).GetAwaiter().GetResult();
```

```
                Thread.Sleep(Timeout.Infinite);
            }
            catch (Exception e)
            {
                ActorEventSource.Current.ActorHostInitializationFailed(e.ToString());
                throw;
            }
        }
    }
}
```

D2 – EmployeesActor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Actors;
using Microsoft.ServiceFabric.Actors.Runtime;
using Microsoft.ServiceFabric.Actors.Client;
using MarcActors.Interfaces;
using Api.Models.Payroll;
using Api.Models.Helpers;
using Api.Data.SqlRepositories.Payroll;
using Api.Data.Models;
using Api.Data.SqlRepositories.Common;
using Api.Models.Common.Providers;
using Api.Logic.Authorization;

namespace MarcActors
{

    [StatePersistence(StatePersistence.None)]
    internal class EmployeesActor : Actor, IEmployeesActor
    {
        private EmployeesActorId _actorId;
        private List<Employee> _employees;
        private string _systemName;
        private SqlEmployeeRepository _repo;
        private QueryOptions _options;

        public EmployeesActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            _actorId = EmployeesActorId.Parse(actorId);
            _systemName = _actorId.SystemName;
        }
```

```
        protected async override Task OnActivateAsync()
        {
            IMetricsContainer metrics = new MetricsContainer();
            LibLog.Logging.ILog logger = null;
            string currentSystem = _systemName; // GET FROM NAME;
            Api.Data.Session.IDbSession session = new ActorDbSession(currentSystem);
            SqlEmployeePropertiesRepository empPrepo = new
SqlEmployeePropertiesRepository(session,logger);


            SystemNameProvider systemNameProvider = new SystemNameProvider(currentSystem);
            SqlCompanyInformationRepository companyInformationRepository = new
SqlCompanyInformationRepository(session, logger, metrics, systemNameProvider);


            _repo = new SqlEmployeeRepository(session, empPrepo, logger,
metrics,systemNameProvider,companyInformationRepository);

            _options = new QueryOptions() { };
            var emp = await _repo.GetAsync(_options);
          _employees = emp.ToList();

            await base.OnActivateAsync();
        }


        public async Task<List<Employee>> GetEmployeesWithOptions(QueryOptions options)
        {
            return _employees;
        }
        public async Task<List<Employee>> GetEmployeesWithEmpOptions(EmployeeQueryOptions
options)
        {
            return _employees;
        }

        private async Task ClearCache(object arg)
        {
            throw new NotImplementedException();
        }

        public async Task<Employee> GetEmployee(long id)
        {

            Employee emp = _employees.FirstOrDefault(e => e.Id == id);
            ActorEventSource.Current.ActorMessage(this, "Returning emp" +emp.ToString());
            return emp;
        }
```

```
        public Task<List<long>> GetEmployeeIds()
        {
            throw new NotImplementedException();
        }

        public Task<List<int>> GetEmployeesAsync()
        {
            throw new NotImplementedException();
        }

        public async Task<List<Employee>> GetEmployeesOnly()
        {
            ActorEventSource.Current.ActorMessage(this, "Returning employees");
            return _employees;
        }

        public Task<List<Employee>> GetEmployeesWithEmployments()
        {
            throw new NotImplementedException();
        }

        public Task<List<int>> UpdateAndGetEmployeesAsync()
        {
            throw new NotImplementedException();
        }
        public async Task<int> TestFunc(string s)
        {
            ActorEventSource.Current.ActorMessage(this, s);
            return 1;
        }

        public async Task<List<Employee>> GetEmployees(List<long> ids)
        {
            return _employees.Where(e => ids.Contains(e.Id)).ToList();
        }
    }
}
```

D3 - LockActor

```
using MarcActors.Interfaces;
using Microsoft.ServiceFabric.Actors.Runtime;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Api.Models.Common.System;
```

```
using Microsoft.ServiceFabric.Actors;
using System.Threading;
using Api.Data.Models;
using Api.Models.Common.DateAndTime;
using Api.Models.Payroll;
using Api.Data.SqlRepositories.Payroll;
using Api.Models.Helpers;

namespace MarcActors
{
    [StatePersistence(StatePersistence.None)]
    internal class LockActor : Actor, ILockActor
    {
        private LockActorId _actorId;
        private DateTime _dateTime;
        private SqlPayrollRunLockedDateRepository _repo;
        private string _systemName;
        private List<PayrollRunLockedDate> _lockedDateList;
        private QueryOptions _lastOptions;

        public LockActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            _actorId = LockActorId.Parse(actorId);
            _systemName = _actorId.SystemName;
        }

        protected override async Task OnActivateAsync()
        {
            DateTime test = new DateTime(2017, 11, 7);
            _dateTime = test;
            IMetricsContainer metrics = new MetricsContainer();
            LibLog.Logging.ILog logger = null;
            string currentSystem = _systemName;

            Api.Data.Session.IDbSession session = new ActorDbSession(currentSystem);

            _repo = new SqlPayrollRunLockedDateRepository(session,logger,metrics);

            var options = new QueryOptions();
            var temp = await _repo.GetAsync(options);
            _lockedDateList = temp.ToList();
            await base.OnActivateAsync();
        }
        public async Task<List<PayrollRunLockedDate>> GetAsync(QueryOptions options)
        {
            //Return only lockedDates matching the QueryOptions
            ActorEventSource.Current.ActorMessage(this, "Returning saved lockesDates");
            return _lockedDateList;
```

```
        }

        public async Task<List<Date>> GetPayrollRunLockedDates()
        {
            var lockedDatesInInterval = (_lockedDateList)
                .Select(p => p.PayrollRunDate)
                .ToList();
            ActorEventSource.Current.ActorMessage(this, "Returning saved lockesDates from
GetPayrollRunLockedDates()");
            return lockedDatesInInterval;
        }

        public async Task<DateTime> GetLockDate()
        {
            return _dateTime;
        }

        public Task<DateTime> SetLockDate()
        {
            throw new NotImplementedException();
        }

        public async Task<int> TestGet(string s)
        {
            ActorEventSource.Current.ActorMessage(this, s);
            return 1;
        }


    }
}
```

D4 – NewsActor

```
using MarcActors.Interfaces;
using Microsoft.ServiceFabric.Actors.Runtime;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Actors;
using System.Net.Http;
using Newtonsoft.Json;
using Microsoft.ServiceFabric.Actors.Client;

namespace MarcActors
{
```

```
    [StatePersistence(StatePersistence.None)]
    internal class NewsActor : Actor, INewsActor
    {
        private const string baseUrl = @"http://www.caspeco.se";
        private NewsActorId _actorId;
        private NewsContainer _newsRecords;
        private IActorTimer _clearCacheTimer;

        public NewsActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            _actorId = NewsActorId.Parse(actorId);
        }

        protected override Task OnActivateAsync()
        {
            _clearCacheTimer = RegisterTimer(
                ClearCache,                     // Callback method
                null,                           // Parameter to pass to the callback method
                TimeSpan.FromMinutes(10),  // Amount of time to delay before the callback is
invoked
                TimeSpan.FromMinutes(10)); // Time interval between invocations of the callback
method

            return base.OnActivateAsync();
        }

        private async Task ClearCache(object arg)
        {
            ActorEventSource.Current.ActorMessage(this, "Clearing cache");
            _newsRecords = null;
        }

        protected override Task OnDeactivateAsync()
        {
            if (_clearCacheTimer != null)
            {
                UnregisterTimer(_clearCacheTimer);
            }

            return base.OnDeactivateAsync();
        }

        private async Task<NewsContainer> ReadNewsRecords(string language)
        {
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri(baseUrl);
```

```
                var response = await
client.GetAsync($"/?json=get_recent_posts&lang={language}");
                response.EnsureSuccessStatusCode();
                string responseBody = await response.Content.ReadAsStringAsync();
                var data = JsonConvert.DeserializeObject<NewsContainer>(responseBody);
                return data;
            };
        }

        public async Task<List<NewsRecord>> GetNews(string[] categories)
        {
            NewsContainer news;

            if (_newsRecords == null)
            {
                _newsRecords = await ReadNewsRecords(_actorId.Language);
            }

            if (_newsRecords.count == 0 || _newsRecords.posts == null)
            {
                return new List<NewsRecord>();
            }

            var result = _newsRecords.posts.Where(post => post.categories.Any(category =>
categories.Any(queryCategory => queryCategory == category.slug))).ToList();
            ActorEventSource.Current.ActorMessage(this, $"Returning {result.Count()} posts");

            return result;
        }

        public async Task<int> TestFunc()
        {
            return 1;
        }
    }
}
```

D5 – PayrollActor

```
using MarcActors.Interfaces;
using Microsoft.ServiceFabric.Actors.Runtime;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Api.Models.Common.System;
using Microsoft.ServiceFabric.Actors;
```

```
using System.Threading;
using PayrollSystemEngine;
using PayrollSystemEngine.PayrollEngine;
using Microsoft.ServiceFabric.Actors.Client;
using Api.Models.Common.DateAndTime;
using Api.Models.Payroll;
using Api.Models.Time;
using Api.Data.Models;
using Api.Models.Helpers;
using Api.Models.Payroll.PayrollRun;
using PayrollSystemEngine.Models;
using PayrollSystemEngine.Utils;
using Api.Models.Common.System.ConfigValue;
using Api.Models.Common.Contract;
using Api.Models.Bookkeeping;
using Api.Models.Common.Calendar;
using Api.Data.SqlRepositories.Payroll;
using Api.Models.Common;
using PayrollSystemEngine.Contracts.SE;
using PayrollSystemEngine.Contracts.NO;
using PayrollSystemEngine.Contracts.DK;
using Api.Logic.Authorization;
using Api.Models.Extensions;
using Api.Logic.Services.Payroll;
using Api.Logic.Services.Common;
using Api.Data.SqlRepositories.Common;

namespace MarcActors
{
    [StatePersistence(StatePersistence.None)]
    internal class PayrollActor : Actor, IPayrollActor
    {
        private PayrollActorId _actorId;
        private PayrollRun _run;
        private PayrollRun _retroDiffs;
        private string _systemName;
        private long _employeeId;
        private Date _payDate;
        private Date _latestLock;

        private List<Date> lockdates;

        private SqlPayrollRunEmployeeSettingRepository _payrollRunEmployeeSettingRepo;

        private string systemCountryIsoCode;

        private Employee _emp;
        private List<Station> _stations;
```

```
        private Dictionary<int, IPayrollActor> _prevPayrollActors;

        private SystemPayrollInformation systemPayrollInformation;
        private PayrollRunDateCollection detailsService;
        private PayDateInfo payDateInfo;
        IMetricsContainer metrics;
        private PayrollRunSettings payrollRunSettings;
        PayrollRunMode mode = PayrollRunMode.Calculate;
        PayrollRunnerOverrides runnerOverrides;


        private IInstancesActor _insActor;
        private IEmployeesActor _empActor;
        private IWorkingShiftsActor _wsActor;
        private ISystemSettingsActor _ssActor;
        private ILockActor _lockActor;
        private IStationsActor _stationActor;

        private List<PayrollArticleInstance> instances;
        private List<WorkingShift> shifts;
        private object settings;
        private SalaryPeriodService _salaryPeriodService;


        public PayrollActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            _actorId = PayrollActorId.Parse(actorId);
            _systemName = _actorId.SystemName;
            _employeeId = _actorId.EmployeeId;
            _payDate = _actorId.PayDate;
        }

        protected async override Task OnActivateAsync()
        {
            ActorEventSource.Current.ActorMessage(this, "Payroll activated" );
            _insActor = ActorProxy.Create<IInstancesActor>(new
InstancesActorId(_systemName,_employeeId).Id);
            _empActor = ActorProxy.Create<IEmployeesActor>(new
EmployeesActorId(_systemName).Id);
            _wsActor = ActorProxy.Create<IWorkingShiftsActor>(new
WorkingShiftsActorId(_systemName).Id);
            _ssActor = ActorProxy.Create<ISystemSettingsActor>(new
SystemSettingsActorId(_systemName).Id);
            _lockActor = ActorProxy.Create<ILockActor>(new LockActorId(_systemName).Id);
            _stationActor = ActorProxy.Create<IStationsActor>(new
StationsActorId(_systemName).GetActorId());
            _prevPayrollActors = new Dictionary<int, IPayrollActor>();
```

```
            Api.Data.Session.IDbSession session = new ActorDbSession(_systemName);

            var insTask = _insActor.GetInstances();
            var wsTask = _wsActor.GetForEmployee(_employeeId);
            var empTask =  _empActor.GetEmployee(_employeeId);
            var stationsTask = _stationActor.GetAllStations();

            await Task.WhenAll(insTask,wsTask, empTask,stationsTask);

            instances = insTask.Result;
            shifts = wsTask.Result;

            _emp = empTask.Result;
            _stations = stationsTask.Result;

            metrics = new MetricsContainer();
            LibLog.Logging.ILog logger = null;

            lockdates = _lockActor.GetPayrollRunLockedDates().Result;
            _latestLock = lockdates.Max();

            Station rootStation = StationHelpers.GetRootStation(_stations);

            systemCountryIsoCode = "se";

            var contractRepo = new SqlContractRepository(session, logger, metrics, _stations,
null);
            var qo = new QueryOptions();
            var contracts = await contractRepo.GetAsync(qo);
            var contractTree = ContractHelpers.GetContractTree(contracts);

            ConfigValueStore configValueStore = contractTree.GetDefaultConfigValues();
            BookkeepingSettings bookkeepingSettings = new BookkeepingSettings();
            var calendars = new CalendarCollection();

            if (systemCountryIsoCode == Country.Sweden.IsoCode)
            {
                calendars.AddCalendar(new SwedishCalendar());
            }
            else if (systemCountryIsoCode == Country.Norway.IsoCode)
            {
                calendars.AddCalendar(new NorwegianCalendar());
            }
            else if (systemCountryIsoCode == Country.Denmark.IsoCode)
            {
                calendars.AddCalendar(new DanishCalendar());
            }
            else
            {
```

```
                // Fallback
                calendars.AddCalendar(new SwedishCalendar());
            }
            var payDateCalendarId =
calendars.GetCalendarForCountry(systemCountryIsoCode).GetCalendarId();
            _payrollRunEmployeeSettingRepo = new SqlPayrollRunEmployeeSettingRepository(session,
logger, metrics);
            var payrollRunEmployeeSettingsTemp = _payrollRunEmployeeSettingRepo.GetAsync(new
QueryOptions());
            List<PayrollRunEmployeeSetting> payrollRunEmployeeSettings =
payrollRunEmployeeSettingsTemp.Result.ToList();

            systemPayrollInformation = new SystemPayrollInformation(
            rootStation,
            configValueStore,
            contractTree,
            bookkeepingSettings,
            calendars,
            systemCountryIsoCode,
            payDateCalendarId,
            lockdates,
            payrollRunEmployeeSettings);

            detailsService = new PayrollRunDateCollection(new
PayrollRunDateCollection(),systemPayrollInformation);
            payrollRunSettings = new PayrollRunSettings(payrollRunEmployeeSettings);

            runnerOverrides = null;

            _salaryPeriodService = new SalaryPeriodService();


            _latestLock = lockdates.Max();

            payDateInfo = new PayDateInfo(_payDate, _latestLock);
            DateTime pd = DateTime.Parse(_payDate.ToString());
            DateTime ll = DateTime.Parse(_latestLock.ToString());
            int dateResult = (DateTime.Compare(pd, ll));
            bool dateBefore = 0 <= dateResult;
            //if not locked, run for retro

            if (dateBefore)
            {
                await DoRetroRuns();
            }
            else
            {
                _run = getFromdb(_payDate);
            }
```

```
        }

        private async Task<Tuple<PayrollRun, PayrollRun>> DoRetroRuns()
        {
            var startRetroRunFrom = StartRetroFromPayDateForEmployee(_emp,
systemPayrollInformation);
            if (startRetroRunFrom.HasValue && startRetroRunFrom.Value.StartOfMonth() <=
_payDate)
            {
                Tuple<PayrollRun, PayrollRun> retro = DoCalcRun(_payDate).Result;
                Tuple<PayrollRun, PayrollRun> runResult = new Tuple<PayrollRun,
PayrollRun>(_run, retro.Item2);
                _retroDiffs = runResult.Item2;
                ;
            }

            return (new Tuple<PayrollRun, PayrollRun>(_run, _retroDiffs));
        }

            public Task<object> GetAccumulatorTransactions()
        {
            throw new NotImplementedException();
        }

        public async Task<Tuple<PayrollRun, PayrollRun>> GetPayrollRun()
        {
            ActorEventSource.Current.ActorMessage(this, _employeeId.ToString() +
"GetPayrollRun()");

            if (_run != null)
            {
                ActorEventSource.Current.ActorMessage(this, "returning _run from " +
_employeeId.ToString() + _payDate.ToString());
                return (new Tuple<PayrollRun, PayrollRun>(_run, _retroDiffs));

            }
            else
            {

                ActorEventSource.Current.ActorMessage(this, "paydate " + _payDate.ToString());


            }
            ActorEventSource.Current.ActorMessage(this, "return (new Tuple<PayrollRun,
PayrollRun>(_run, null));");
            return (new Tuple<PayrollRun, PayrollRun>(_run, null));

        }
```

```
        private Date? StartRetroFromPayDateForEmployee(Employee employee,
SystemPayrollInformation systemPayrollInformation)
        {
            return employee.Employments.Min(employment =>
StartRetroFromPayDateForEmployment(employment, systemPayrollInformation));
        }

        private Date? StartRetroFromPayDateForEmployment(Employment employment,
SystemPayrollInformation systemPayrollInformation)
        {
            return employment.PayrollsValidUntil == null
                ? (Date?)null
                :
_salaryPeriodService.GetPayDateFromDateInSalaryPeriod(systemPayrollInformation,
employment.PayrollsValidUntil.Value, null);
        }

        private PayrollRun getFromdb(Date date, PayrollRun previusRun =null)
        {
            _run = CloneEngine.Run(_emp, instances, shifts, systemPayrollInformation, date,
detailsService, payDateInfo, metrics, payrollRunSettings, mode, runnerOverrides,
previousPayrollRun: previusRun).Result;
            return _run;
        }

        private async Task<Tuple<PayrollRun, PayrollRun>> DoCalcRun(Date pd)
        {

            var previousMonth = pd.AddMonths(-1);
            Date prevDate = systemPayrollInformation.GetPayDate(previousMonth.Year,
previousMonth.Month);

            _prevPayrollActors[previousMonth.Month] = ActorProxy.Create<IPayrollActor>(new
PayrollActorId(_systemName,_employeeId,prevDate).Id);

            Tuple<PayrollRun, PayrollRun> temp = await
_prevPayrollActors[previousMonth.Month].GetPayrollRun();
            PayrollRun diff = null;

            if (temp.Item1 != null)
            {
                detailsService.AddPayrollRun(temp.Item1);
            }
            var calcRun = CloneEngine.Run(_emp, instances, shifts, systemPayrollInformation,
_payDate, detailsService, payDateInfo, metrics, payrollRunSettings, mode,
runnerOverrides).Result;

            return new Tuple<PayrollRun, PayrollRun>(calcRun, diff);
        }
```

```
        public Task<object> GetPayslipRows()
        {
            throw new NotImplementedException();
        }

        public Task<Tuple<int, long>> GetResultPayRollAsync(CancellationToken cancellationToken)
        {
            throw new NotImplementedException();
        }
        public async Task<PayrollRun> TestGet()
        {


            _run = CloneEngine.Run(_emp, instances, shifts, systemPayrollInformation, _payDate,
detailsService, payDateInfo, metrics, payrollRunSettings, mode, runnerOverrides).Result;

            return _run;
        }

        public async Task Test(Employee employee1,
            List<PayrollArticleInstance> payrollArticleInstances1,
            List<WorkingShift> workingShifts1,
            SystemPayrollInformation systemPayrollInformation1,
            PayDateInfo payDateInfo1,
            PayrollRunSettings payrollRunSettings1,
            PayrollRunMode mode1,
            PayrollRunnerOverrides runnerOverrides1)
        {
            ActorEventSource.Current.ActorMessage(this,"!");
            var calcRun =
CloneEngine.Run(employee1,payrollArticleInstances1,workingShifts1,systemPayrollInformation1,_pay
Date,detailsService,payDateInfo1,metrics,payrollRunSettings1,mode1,runnerOverrides1).Result;
        }

        private PayrollRun calcDiff(PayrollRun current,PayrollRun prev)
        {
            var diff = PayrollRunDiffCalculator.CalculateDiff(prev, current,
PayrollRunDiffCalculatorMode.All, systemPayrollInformation);
            return diff;
        }
    }
}
```

D6 – StationsActor

```
using MarcActors.Interfaces;
using Microsoft.ServiceFabric.Actors.Runtime;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Api.Models.Common.System;
using Microsoft.ServiceFabric.Actors;
using Api.Data.Models;
using Api.Logic.Services;
using Api.Logic.Services.Common;
using Api.Data.Repositories;
using Api.Data.SqlRepositories;
using Api.Models.Helpers;
using Microsoft.Practices.EnterpriseLibrary.TransientFaultHandling;
using Dapper;
using Api.Models.Common.DateAndTime;
using System.Data;
using Api.Data.Dapper;

namespace MarcActors
{
    [StatePersistence(StatePersistence.None)]
    internal class StationsActor : Actor, IStationsActor
    {
        private StationsActorId _actorId;
        private List<Station> _stations;
        private SqlStationRepository _repo;
        private string _systemName;

        public StationsActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            ActorEventSource.Current.ActorMessage(this, "StationsActor Constructor");
            _actorId = StationsActorId.Parse(actorId);
            _systemName = _actorId.SystemName;

        }

        protected override async Task OnActivateAsync()
        {
            // create StationReposistory
            // also need to create/get a DBSession
            // read data from repository

            IMetricsContainer metrics = new MetricsContainer();
```

```
            LibLog.Logging.ILog logger = null;
            string currentSystem = _systemName; // GET FROM NAME;

            Api.Data.Session.IDbSession session = new ActorDbSession(currentSystem);
            _repo = new SqlStationRepository(session, logger, metrics);

            var options = new QueryOptions() { };
            var stations = await _repo.GetAsync(options);
            _stations = stations.ToList();

            await base.OnActivateAsync();
        }

        public async Task<List<Station>> GetAllStations()
        {

            ActorEventSource.Current.ActorMessage(this, "Returning stations");
            return _stations;
        }


        public async Task AddStations(List<Station> stations)
        {
            _stations.AddRange(stations);
        }
        public async Task<List<Station>> GetStations(List<long> stationIds)
        {
            ActorEventSource.Current.ActorMessage(this, "In getStations");
            return _stations.Where(station => stationIds.Contains(station.Id)).ToList();
        }


        public async Task<Station> GetStation(long stationId)
        {
            ActorEventSource.Current.ActorMessage(this, "In getStation");
            List<Station> temp = _stations.Where(station => stationId == station.Id).ToList();
            return temp[0];
        }

        public async Task UpdateStations(List<Station> stations)
        {
            List<long> idList = stations.Select(s => s.Id).ToList();
            _stations = _stations.Except(_stations.Where(s =>
idList.Contains(s.Id)).ToList()).ToList();
            _stations.AddRange(stations);
        }

        public async Task<int> TestFunc(string s)
        {
```

```
            ActorEventSource.Current.ActorMessage(this, s);
            return 1;
        }

    }
}
```

D7 – WorkingShiftsActor

```
using MarcActors.Interfaces;
using Microsoft.ServiceFabric.Actors.Runtime;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Api.Models.Common.System;
using Microsoft.ServiceFabric.Actors;
using System.Threading;
using Api.Models.Time;
using Api.Data.Models;
using Api.Data.SqlRepositories;
using Api.Models.Helpers;

namespace MarcActors
{
    [StatePersistence(StatePersistence.None)]
    internal class WorkingShiftsActor : Actor, IWorkingShiftsActor
    {
        private WorkingShiftsActorId _actorId;
        private List<WorkingShift> _workingShiftList;
        private string _systemName;
        private SqlWorkingShiftRepository _repo;

        public WorkingShiftsActor(ActorService actorService, ActorId actorId)
            : base(actorService, actorId)
        {
            _actorId = WorkingShiftsActorId.Parse(actorId);
            _systemName = _actorId.SystemName;
        }
        protected override async Task OnActivateAsync()
        {
            IMetricsContainer metrics = new MetricsContainer();
            LibLog.Logging.ILog logger = null;
            string currentSystem = _systemName;

            Api.Data.Session.IDbSession session = new ActorDbSession(currentSystem);
```

```
            _repo = new SqlWorkingShiftRepository(session, logger, metrics);

            var options = new WorkingShiftQueryOptions();
            var temp = await _repo.GetAsync(options);
            _workingShiftList = temp.ToList();
        }

        public async Task<List<WorkingShift>>
GetListWorkingShiftsList(List<WorkingShiftQueryOptions> optionList)
        {
            if (_workingShiftList == null)
            {
                foreach (var queryOptions in optionList)
                {
                    var temp = await _repo.GetAsync(queryOptions);
                    _workingShiftList = temp.Union(temp, new ShiftComparer()).ToList();
                }
            }

            return _workingShiftList;
        }
        public async Task<List<WorkingShift>> GetListWorkingShifts(WorkingShiftQueryOptions
options)
        {

                var temp = await _repo.GetAsync(options);
                _workingShiftList = temp.ToList();

            return _workingShiftList;
        }

        public async Task<List<WorkingShift>> GetListWorkingShiftsWithoutOptions()
        {
            return _workingShiftList;
        }

        public async Task<List<WorkingShift>> GetForEmployee(long employeeId)
        {
            return _workingShiftList.Where(ws => ws.EmployeeId == employeeId).ToList();
        }

        public Task<object> GetWorkingShifts(long empId, DateTime start, DateTime end)
        {
            throw new NotImplementedException();
        }

        public Task<int> TestGet()
        {
            throw new NotImplementedException();
```

```
        }

        public async Task<int> TestFunc(string s)
        {
            ActorEventSource.Current.ActorMessage(this, s);
            return 1;
        }

    }
    class ShiftComparer : IEqualityComparer<WorkingShift>
    {
        public bool Equals(WorkingShift x, WorkingShift y)
        {
            return x.Id == y.Id;
        }

        public int GetHashCode(WorkingShift obj)
        {
            return obj.Id.GetHashCode();
        }
    }
}
```

Appendix E – Namespace IMarcActors från implementation i lönemotorn

E1 – IEmployeesActor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Actors;
using Api.Models.Payroll;
using Api.Data.Models;

namespace MarcActors.Interfaces
{

    public interface IEmployeesActor : IActor
    {
        Task<List<int>> GetEmployeesAsync();

        Task<List<int>> UpdateAndGetEmployeesAsync();
```

```
        Task<List<long>> GetEmployeeIds();

        Task<Employee> GetEmployee(long id);

        Task<List<Employee>> GetEmployeesWithEmployments();

        Task<List<Employee>> GetEmployeesOnly();

        Task<List<Employee>> GetEmployeesWithEmpOptions(EmployeeQueryOptions options);

        Task<List<Employee>> GetEmployeesWithOptions(QueryOptions options);

        Task<List<Employee>> GetEmployees(List<long> ids);

        Task<int> TestFunc(string s);

    }
    public class EmployeesActorId
    {
        public string SystemName;


        public EmployeesActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static EmployeesActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new EmployeesActorId(part);
        }
    }
}
```

E2 – ILockActor

```
using Api.Data.Models;
using Api.Models.Common.DateAndTime;
using Api.Models.Common.System;
using Api.Models.Payroll;
using Microsoft.ServiceFabric.Actors;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MarcActors.Interfaces
{
    public interface ILockActor : IActor
    {
        Task<DateTime> GetLockDate();

        Task<DateTime> SetLockDate();

        Task<int> TestGet(string s);

        Task<List<PayrollRunLockedDate>> GetAsync(QueryOptions options);

        Task<List<Date>> GetPayrollRunLockedDates();
    }
    public class LockActorId
    {
        public string SystemName;

        public LockActorId(string systemName)
        {
            SystemName = systemName;
        }

        public override string ToString()
        {
            return $"{SystemName}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }
```

```
        public static LockActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new LockActorId(part);
        }
    }
}
```

E3 – INewsActor

```
using Microsoft.ServiceFabric.Actors;
using Microsoft.ServiceFabric.Actors.Remoting.FabricTransport;
using Microsoft.ServiceFabric.Services.Remoting;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Text;
using System.Threading.Tasks;

namespace MarcActors.Interfaces
{
    [DataContract]
    public class NewsContainer
    {
        [DataMember]
        public string status;
        [DataMember]
        public int count;
        [DataMember]
        public int count_total;
        [DataMember]
        public int pages;
        [DataMember]
        public NewsRecord[] posts;
    }

    [DataContract]
    public class Author
    {
        [DataMember]
        public string name;
        [DataMember]
        public string first_name;
        [DataMember]
        public string last_name;
    }
```

```
[DataContract]
public class Category
{
    [DataMember]
    public int id;
    [DataMember]
    public string slug;
    [DataMember]
    public string title;
}

[DataContract]
public class NewsRecord
{
    [DataMember]
    public int id;
    [DataMember]
    public string status;
    [DataMember]
    public string slug;
    [DataMember]
    public string title;
    [DataMember]
    public string excerpt;
    [DataMember]
    public DateTime date;
    [DataMember]
    public DateTime modified;
    [DataMember]
    public Category[] categories;
    [DataMember]
    public Author author;
    [DataMember]
    public string content;
}

public class NewsActorId
{
    public string Language;

    public NewsActorId(string language)
    {
        Language = language;
    }

    public ActorId GetActorId()
    {
        return new ActorId(Language);
```

```
        }

        public static NewsActorId Parse(ActorId actorId)
        {
            return new NewsActorId(actorId.GetStringId());
        }
    }

    public interface INewsActor : IActor
    {
        Task<List<NewsRecord>> GetNews(string[] categories);

        Task<int> TestFunc();
    }
}
```

E4 – IPayrollActor

```
using Api.Models.Common.DateAndTime;
using Api.Models.Common.System;
using Microsoft.ServiceFabric.Actors;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using PayrollSystemEngine.PayrollEngine;
using Api.Models.Payroll;
using Api.Models.Time;
using PayrollSystemEngine;
using Api.Models.Helpers;
using Api.Models.Payroll.PayrollRun;
using PayrollSystemEngine.Models;
using Api.Logic.Services.Payroll;

namespace MarcActors.Interfaces
{
    public class PayrollActorId
    {
        public string SystemName;
        public long EmployeeId;
        public Date PayDate;

        public PayrollActorId(string systemName, long employeeId, Date payDate)
        {
            SystemName = systemName;
            EmployeeId = employeeId;
```

```
            PayDate = payDate;
        }

        public override string ToString()
        {
            return $"{SystemName};{EmployeeId};{PayDate}";
        }

        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }

        public static PayrollActorId Parse(ActorId id)
        {
            var parts = id.GetStringId().Split(';');
            return new PayrollActorId(parts[0], long.Parse(parts[1]), Date.Parse(parts[2]));
        }
    }


    public interface IPayrollActor : IActor
    {
        Task<Tuple<int, long>> GetResultPayRollAsync(CancellationToken cancellationToken);

        Task<Tuple<PayrollRun, PayrollRun>> GetPayrollRun();

        Task<object> GetAccumulatorTransactions();

        Task<object> GetPayslipRows();

        Task<PayrollRun> TestGet();

        Task Test(Employee employee,
            List<PayrollArticleInstance> payrollArticleInstances,
            List<WorkingShift> workingShifts,
            SystemPayrollInformation systemPayrollInformation,
            PayDateInfo payDateInfo,
            PayrollRunSettings payrollRunSettings,
            PayrollRunMode mode,
            PayrollRunnerOverrides runnerOverrides);
    }
}
```

E5 – IStationsActor

```
using Api.Models.Common.System;
using Microsoft.ServiceFabric.Actors;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.Serialization;
using Microsoft.ServiceFabric.Actors.Remoting.FabricTransport;
using Microsoft.ServiceFabric.Services.Remoting;
using Api.Models.Common.DateAndTime;
using NodaTime;
using Api.Data.Models;
using Api.Logic.Services;
using Api.Data.Repositories;

namespace MarcActors.Interfaces
{
    public class StationsActorId
    {
        public string SystemName;
        public StationsActorId(string systemName)
        {
            SystemName = systemName;
        }

        public ActorId GetActorId()
        {
            return new ActorId(SystemName);
        }

        public static StationsActorId Parse(ActorId actorId)
        {
            return new StationsActorId(actorId.GetStringId());
        }
    }

    public interface IStationsActor : IActor
    {
        Task<List<Station>> GetStations(List<long> stationIds);
        Task AddStations(List<Station> stations);
        Task UpdateStations(List<Station> stations);
        Task<Station> GetStation(long stationId);
        Task<int> TestFunc(string s);
        Task<List<Station>> GetAllStations();
    }
}
```

E6 – IWorkingShiftsActor

```
using Api.Data.Models;
using Api.Models.Common.System;
using Api.Models.Time;
using Microsoft.ServiceFabric.Actors;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace MarcActors.Interfaces
{
    public interface IWorkingShiftsActor : IActor
    {
        Task<object> GetWorkingShifts(long empId, DateTime start, DateTime end);
        Task<List<WorkingShift>> GetListWorkingShifts(WorkingShiftQueryOptions options);
        Task<List<WorkingShift>> GetListWorkingShiftsList(List<WorkingShiftQueryOptions>
optionList);
        Task<List<WorkingShift>> GetListWorkingShiftsWithoutOptions();
        Task<List<WorkingShift>> GetForEmployee(long employeeId);
        Task<int> TestGet();
        Task<int> TestFunc(string s);
    }
    public class WorkingShiftsActorId
    {
        public string SystemName;
        public WorkingShiftsActorId(string systemName)
        {
            SystemName = systemName;
        }
        public override string ToString()
        {
            return $"{SystemName}";
        }
        public ActorId Id
        {
            get
            {
                return new ActorId(ToString());
            }
        }
        public static WorkingShiftsActorId Parse(ActorId id)
        {
            var part = id.GetStringId();
            return new WorkingShiftsActorId(part);
        }
    }
}
```

Appendix F – Resultattabeller testkörningar

F1

Tabell 1 - Mätning av anrop efter arbetspass utan Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	6553,1 ms	6627,4 ms	7300,8 ms	4732,8 ms
Långsammast anrop	6553,1 ms (Anrop #0)	6627,4 ms (Anrop #0)	7300,8 ms (Anrop #0)	4732,8 ms (Anrop #0)
Näst långsammaste anropet	88,2 ms (anrop #628)	70,0 ms (Anrop #3146)	51,5 ms (Anrop #1329)	53,8 ms (Anrop #4043)
Snabbast anrop	13,2 ms (Anrop #3707)	13,3 ms (Anrop #4618)	13,3 ms (Anrop #9963)	13,3 ms (Anrop #6117)
Genomsnittlig tid	21,1 ms	21,3 ms	19,7 ms	19,2 ms
Genomsnittlig tid bortsett från första anropet	20,5 ms	20,6 ms	19,0 ms	18,7 ms
Total tid för 10 000 anropar	211 987 ms	213 005 ms	197 206 ms	191 825 ms
Variationsbredd utan första anropet	75,0 ms	56,7 ms	38,2 ms	40,5 ms

F2

Tabell 3 - Mätning av anrop efter arbetspass med Service Fabric Reliable Actor

	Test 1_2	Test 2_2	Test 3	Test 4
Första anropet	8361,6 ms	7497,7 ms	8198,7 ms	6545,3 ms
Långsammast anrop	8361,6 ms (Anrop #0)	7497,7 ms (Anrop #0)	8198,7 ms (Anrop #0)	6545,3 ms (Anrop #0)
Näst långsammaste anropet	100,0 ms (Anrop #9416)	61,5 ms (Anrop #4764)	68,0 ms (Anrop #3053)	5399,8 ms (Anrop #9387)
Snabbast anrop	13,4 ms (Anrop #4396)	13,4 ms (Anrop #5621)	13,4 ms (Anrop #5376)	13,3 ms (Anrop #6874)
Genomsnittlig tid	20,0 ms	18,9 ms	19,3 ms	20,0 ms
Genomsnittlig tid bortsett från första anropet	19,1 ms	18,9 ms	18,5 ms	19,1 ms
Total tid för 10 000 anropar	199 888 ms	189 498 ms	193 690 ms	197 744 ms
Variationsbredd utan första anropet	86,6 ms	48,1 ms	54,6 ms	5386,5 ms

F3

Tabell 5 - Mätning av anrop efter anställda utan Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	3486,8 ms	5581,8 ms	11 361,7 ms	1322,0 ms
Långsammast anrop	3486,8 ms (Anrop #0)	5581,8 ms (Anrop #0)	11 361,7 ms (Anrop #0)	1322,0 ms (Anrop #0)
Näst långsammaste anropet	313,8 ms (Anrop #6929)	185,5 ms (Anrop #1712)	395,4 ms (Anrop #12)	316,1 ms (Anrop #9151)
Snabbast anrop	20,3 ms	20,8 ms	20,9 ms	21,0 ms

	(Anrop #9742)	(Anrop #6314)	(Anrop #3481)	(Anrop #897)
Genomsnittlig tid	25,3 ms	25,1 ms	26,6 ms	26,0 ms
Genomsnittlig tid bortsett från första anropet	25,0 ms	24,6 ms	25,5 ms	25,9 ms
Total tid för 10 000 anropar	253 654 ms	251 600 ms	266 197 ms	260 197 ms
Variationsbredd utan första anropet	293,5 ms	164,7 ms	374,5 ms	295,1 ms

F4

Tabell 7 - Mätning av anrop efter anställda med Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	17 411,5 ms	13 483,5 ms	7276,4 ms	17 551,0 ms
Långsammast anrop	17 411,5 ms (Anrop #0)	13 483,5 ms (Anrop #0)	7276,4 ms (Anrop #0)	17 551,0 ms (Anrop #0)
Näst långsammaste anropet	188,0 ms (Anrop #1740)	129,5 ms (Anrop #3003)	147,8 ms (Anrop #4712)	49,5 ms (Anrop #5561)
Snabbast anrop	15,9 ms (Anrop #686)	16,2 ms (Anrop #7651)	16,2 ms (Anrop #9638)	16,1 ms (Anrop #2606)
Genomsnittlig tid	23,7 ms	22,4 ms	19,6 ms	19,9 ms
Genomsnittlig tid bortsett från första anropet	21,9 ms	21,0 ms	18,8 ms	18,2 ms
Total tid för 10 000 anropar	237 039 ms	223 818 ms	195 776 ms	199 518 ms
Variationsbredd utan första anropet	172,1 ms	113,3 ms	131,6 ms	33,4 ms

F5

Tabell 9 - Mätning av anrop efter låsta datum utan Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	4534,7 ms	9346,7 ms	4196,0 ms	16 720,7 ms
Långsammast anrop	4534,7 ms (Anrop #0)	9346,7 ms (Anrop #0)	4196,0 ms (Anrop #0)	16 720,7 ms (Anrop #0)
Näst långsammaste anropet	361,4 ms (Anrop #1)	66,7 ms (Anrop #8954)	442,7 ms (Anrop #1663)	82,6 ms (Anrop #8618)
Snabbast anrop	13,4 ms (Anrop #9202)	13,4 ms (Anrop #3676)	13,3 ms (Anrop #3031)	13,3 ms (Anrop #8721)
Genomsnittlig tid	16,3 ms	16,8 ms	16,5 ms	16,7 ms
Genomsnittlig tid bortsett från första anropet	15,9 ms	15,8 ms	16,0 ms	15,0 ms
Total tid för 10 000 anropar	163 665 ms	167 835 ms	164 757 ms	167 292 ms
Variationsbredd utan första anropet	348,0 ms	53,3 ms	429,4 ms	69,3 ms

F6

Tabell 11 - Mätning av anrop efter låsta datum med Service Fabric Reliable Actor

	Test 1_2	Test 2_2	Test 3_2	Test 4_2
Första anropet	2007,8 ms	1401,1 ms	2871,5 ms	3649,9 ms
Långsammast anrop	2007,8 ms (Anrop #0)	1401,1 ms (Anrop #0)	2871,5 ms (Anrop #0)	3649,9 ms (Anrop #0)
Näst långsammaste anropet	79,2 ms (Anrop #3572)	160,9 ms (Anrop #9834)	157,7 (Anrop #178)	85,8 ms (Anrop #2320)
Snabbast anrop	12,1 ms (Anrop	12,1 ms (Anrop	12,1 ms (Anrop	12,0 ms (Anrop

	#1136)	#3656)	#6255)	#7900
Genomsnittlig tid	14,1 ms	13,9 ms	14,2 ms	14,4 ms
Genomsnittlig tid bortsett från första anropet	13,9 ms	13,8 ms	13,9 ms	14,0 ms
Total tid för 10 000 anropar	141 528 ms	139 343 ms	142 398 ms	143 978 ms
Variationsbredd utan första anropet	67,1 ms	148,8 ms	145,6 ms	73,8 ms

F7

Tabell 13 - Mätning av anrop efter stationer utan Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	85,9 ms	62,0 ms	88,9 ms	48,0 ms
Långsammast anrop	85,9 ms (Anrop #0)	62,0 ms (Anrop #0)	88,9 ms (Anrop #0)	48,0 ms (Anrop #0)
Näst långsammaste anropet	43,2 ms (Anrop #236)	40,5 ms (Anrop #174)	38,6 ms (Anrop #9767)	38,6 ms (Anrop #9522)
Snabbast anrop	10,1 ms (Anrop #8751)	10,1 ms (Anrop 5755)	10,1 ms (Anrop #1069)	10,1 ms (Anrop #70)
Genomsnittlig tid	10,9 ms	10,7 ms	10,7 ms	10,8 ms
Genomsnittlig tid bortsett från första anropet	10,9 ms	10,7 ms	10,7 ms	10,8 ms
Total tid för 10 000 anropar	108 905 ms	107 236 ms	107 303 ms	108 014 ms
Variationsbredd utan första anropet	33,1 ms	30,4 ms	28,5 ms	28,5 ms

F8

Tabell 15 – Mätning av anrop efter stationer med Service Fabric Reliable Actor

	Test 1	Test 2	Test 3	Test 4
Första anropet	5329,1 ms	5531,9 ms	3754,9 ms	3294,5 ms
Långsammaste anrop	5329,1 ms (Anrop #0)	5531,9 ms (Anrop #0)	3754,9 ms (Anrop #0)	3294,5 ms (Anrop #0)
Näst långsammaste anropet	218,2 ms (Anrop 318)	261,4 ms (Anrop #5819)	120,1 ms (Anrop #1649)	83,9 ms (Anrop #8300)
Snabbast anrop	12,4 ms (Anrop #1522)	12,4 ms (Anrop #4763)	12,5 ms (Anrop #2425)	12,4 ms (Anrop #4002)
Genomsnittlig tid	15,9 ms	15,3 ms	15,1 ms	15,4 ms
Genomsnittlig tid bortsett från första anropet	15,4 ms	14,8 ms	14,8 ms	15,0 ms
Total tid för 10 000 anropar	159 029 ms	153 381 ms	151 517 ms	153 732 ms
Variationsbredd utan första anropet	205,8 ms	249,0 ms	107,6 ms	71,5 ms