SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Praćenje aktivnosti na društvenim mrežama pomoću reaktivnih agenata

Seminarski rad

Nikola Majcen
br. indeksa: 44445-15/R

Mentor:
Doc. dr. sc. Markus Schatten

Varaždin, 4. siječnja 2017.

Sadržaj

Poglavlje 1

Uvod

Tema ovog seminarskog rada je praćenje aktivnosti na društvenim mrežama pomoću reaktivnih agenata, odnosno izrada jednostavne aplikacije gdje se pomoću agenata mogu pratiti objave na društvenim mrežama u realnom vremenu.

Seminarski rad uključuje teorijsku obradu agenata, ali praktični prikaz i implementaciju rješenja koje je vezano za praćenje aktivnosti na društvenim mrežama Twitter i Facebook. Praćenje na društvenim mrežama uključuje praćenje statusa i/ili hashtag-ova. Rad je podjeljen na cjeline i to:

- Opis zadatka
- Implementacija rješenja
- Demonstracija rješenja

Poglavlje 2

Opis zadatka

Tema ovog seminarskog rada jest izrada aplikacije koja korisniku omogućuje da pretražuje određene društvene mreže preko hashtag-ova ili statusa, na način da se na iste pretplati te obrada literature vezane za reaktivne agente koji se koriste u samoj aplikaciji.

2.1 Koncept zadatka

Da bi korisnik mogao pretraživati određene društvene mreže, za početak je potrebno pretpostaviti koji će biti elementi pretrage i kako će se moći podaci pratiti.

U ovom zadatku će se koristiti društvene mreže Facebook i Twitter, gdje će se kod svake od njih pratiti određena aktivnost korisnika. Zbog određenih mjera privatnosti, nije bilo moguće izvesti iste funkcionalnosti za svaku od društvenih mreža, pa je shodno tome omogućeno sljedeće; za društvenu mrežu Facebook omogućeno je:

- Praćenje statusa korisnika čiji su profili javni

dok je za društvenu mrežu Twitter omogućeno sljedeće:

- Praćenje statusa (tweet) svih korisnika (korisnički profil ne mora biti javan)
- Praćenje statusa (tweet) koji sadrže određeni hashtag

Pretplata na promjenu statusa ili praćenje statusa sa određenim hashtag-om odvija se na način da korisnik definira vrijeme u sekundama u kojim će intervalima i kolko dugo pratiti promjene na određenoj društvenoj mreži. Detaljnije informacije o implementaciji zadatka i potrebnim dodatnih podacima za spajanje na određene društvene mreže nalaze se u sljedećem poglavlju.

Za implementaciju rješenja ovog zadatka koristio sam višeagentni sustav koji se sastoji od dva, odnosno tri reaktivna agenta. Dva reaktivna agenta omogućuju pretraživanje i praćenje promjena na društvenim mrežama, a skupljene podatke šalju trećem agentu koji dohvaća sve promjene svih agenata. Za svaku društvenu mrežu definiran je posebni agent koji koristi specifične načine dohvata podataka za svaku od društvenih mreža. Nakon što određeni agent za dohvaćanje podataka s društvene mreže završi s radom, treći agent (koji sakuplja podatke) šalje izvještaj u obliku svih sakupljenih podataka koje mu je taj agent poslao. Na taj način završava praćenje i izvještavanje za tog agenta, dok svi ostali nastavljaju s radom.

2.2 Prikaz znanstvenih članaka ili neki naziv cool :) teorijsko nešto :D

Poglavlje 3

Implementacija rješenja

U ovom poglavlju prikazat ću implementaciju rješenja za praćenje na društvenim mrežama. Za implementaciju je korišten programski jezik Python verzije 2.7. Kao biblioteka koja omogućuje rad s agentima korišten je SPADE.

Za početak rada, potrebno je u izvornom direktoriju otvoriti datoteku *config.json* te unutar nje definirati naziv i lozinku agenta izvjestitelja. Zadano ime agenta je *reporter@127.0.0.1*, dok je lozinka *secret*.

Uz tu datoteku, postoji i datoteka *credentials-sample.json*, koja predstavlja podatke koji su potrebni da bi se aplikacija mogla povezati sa određenom društvenom mrežom. Detaljnije informacije za unos podataka za svaku od društvenih mreža bit će prikazane u zasebnom poglavlju za svakog agenta posebno, ali prije korištenja potrebno je istu preimenovati u *credentials.json*.

3.1 Implementacija Facebook agenta

Prije detaljnog objašnjavanja Facebook agenta, potrebno je prvo definirati potrebne podatke kako bi se aplikacija mogla povezati sa Facebook servisima. Za to je potrebno otvoriti datoteku *credentials.json* te unutar objekta `"facebook"` za ključ `"access_token"` unijeti vrijednost tokena koja se dobije kada se kreira nova aplikacija unutar Facebook-ovog sučelja za razvijatelje aplikacija. Bez vrijednosti `"access_token"`-a nije moguće koristiti Facebook agenta.

Prikaz implementacije podijeljen je na tri dijela, a prikazuju sljedeće:

- implementaciju modela,
- implementaciju API-a
- implementaciju reaktivnog agenta

3.1.1 Prikaz implementacije modela

Kako bi se dohvatili podaci za spajanje na Facebook-ov servis, kreiran je model *FacebookCredentials*, a sadrži sljedeće:

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-
```

```
import json


class FacebookCredentials:

    def __init__(self, filename):
        self.filename = filename

    def get_credentials(self):
        json_data = self.__read_configuration_file()
        api_key = json_data["facebook"]["access_token"]
        return api_key

    def __read_configuration_file(self):
        file = open(self.filename, "r")
        data = json.load(file)
        file.close()
        return data
```

Model sadrži dvije metode; *get_credentials()* za dohvat podataka za spajanje na servis, te *__read_configuration_file()* koja omogućuje čitanje datoteke u kojem se nalaze podaci za spajanje na servis.

Sljedeći modeli su *FacebookUser* i *FacebookStatus*:

```
# ! /usr/bin/env python
# -*- coding: utf-8 -*-

from datetime import datetime


class FacebookUser:

    def __init__(self, data):
        self.id = data["id"]
        self.username = data["username"]
        self.name = data["name"]
        city = data["location"]["city"]
        country = data["location"]["country"]
        state = data["location"]["state"]
        self.location = "{}, {}, {}".format(city, country, state)


class FacebookStatus:

    def __init__(self, data):
        self.name = None
        self.username = None
```

```
        self.status_type = data["status_type"]
        self.date = datetime.strptime(data["updated_time"], "%Y-%m-%dT%H:%M:%S+0000")
        if "message" in data:
            self.message = data["message"].strip()
        else:
            self.message = ""

        if "link" in data:
            self.link = data["link"]
        else:
            self.link = ""
```

Klasa *FacebookUser* definira osnovne informacije o korisniku, kao što je njegov id, korisničko ime, ime i prezime te lokacija. Klasa *FacebookStatus* definira osnovne informacije o statusu, a sadrži ime i prezime korisnika koji je objavio status, korisničko ime, tip statusa, vrijeme objave, te ovisno o tome da li se radi o tekstualnom statusu ili slici poruku odnosno poveznicu.

3.1.2 Prikaz implementacije API-a

Sljedeću stvar koju sam kreirao je Facebook API koji zapravo definira pozive metoda koje se spajaju na Facebook-ov službeni API te na temelju tih poziva se dobivaju određeni podaci koje korisnik traži.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import json

import facebook

from social.facebook.facebook_models import *


class FacebookAPI:

    def __init__(self, access_token):
        self.access_token = access_token

    def search_posts(self, username):
        user_id, user_username, user_name = self.__fetch_user_details(username)
        if user_id is None:
            print "Token is expired."
            return None

        print user_id
        graph = facebook.GraphAPI(access_token=self.access_token)
        batched_requests = '[{"method": "GET", "relative_url": "' + user_id + '/feed"
```

```
        try:
            posts_data = graph.request("", post_args={"batch": batched_requests})
        except:
            print "Token has expired."

        for key, value in posts_data[0].items():
            if key == "body":
                json_posts = json.loads(value)["data"]
                break

        posts = []
        for json_post in json_posts:
            post = FacebookStatus(json_post)
            post.username = user_username
            post.name = user_name
            posts.append(post)
        return posts

    def __fetch_user_details(self, username):
        graph = facebook.GraphAPI(access_token=self.access_token)
        batched_requests = '[ {"method": "GET", "relative_url": "  "' + username + '"} ]'
        try:
            user_data = graph.request("", post_args={"batch": batched_requests})
        except:
            return None

        if user_data is None:
            return None

        for key, value in user_data[0].items():
            user = FacebookUser(json.loads(value))
            break
        return user.id, user.username, user.name
```

Klasa *FacebookAPI* definira poziv za dohvaćanje novih statusa preko metode `search_posts(username)`. Ova metoda na temelju korisničkog id-a traži nove statuse ili slike korisnika, te iste dohvaća i vraća kao rezultat. Prethodno navedena metoda mora obavezno imati id korisničkog računa potrebnog za dohvaćanje podataka, a kako bi se isti dobio, potrebno je pomoću metode `__fetch_user_details(username)` dohvatiti korisnikov id.

3.1.3 Prikaz implementacije reaktivnog agenta

Ovdje ću prikazati konačnu implementaciju agenta koji na Facebook društvenoj mreži dohvaća promjene u smislu novih statusa koji uključuju tekst ili slike.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-
```

```
import json
from datetime import datetime, timedelta

from spade import Agent
from spade import Behaviour
from spade import ACLMessage
from spade import AID

from social.facebook.facebook_api import FacebookAPI
from social.facebook.facebook_credentials import FacebookCredentials
from social.shared.report_message import ReportMessage


class FacebookAgent(Agent.Agent):

    class FetchBehaviour(Behaviour.PeriodicBehaviour):

        def __init__(self, time, period, credentials_filename):
            Behaviour.PeriodicBehaviour.__init__(self, period)
            self.periods = time / period
            self.current_date = datetime.now()
            facebook_credentials = FacebookCredentials(credentials_filename)
            self.facebook_api = FacebookAPI(facebook_credentials.get_credentials())

        def onStart(self):
            print "[" + self.myAgent.getName() + "] Status activity fetch started."
            self.__send_registration_message()

        def onEnd(self):
            print "[" + self.myAgent.getName() + "] Status activity fetch ended."
            self.__send_report_message()

        def _onTick(self):
            if self.periods > 0:
                self.periods -= 1
                self.fetch_data()
            else:
                self.kill()

        def fetch_data(self):
            print "[" + self.myAgent.getName() + "] Fetching statuses... (No. of peric
            results = self.facebook_api.search_posts(self.myAgent.keyword)

            statuses_found = 0
```

```
            for status in results:
                if (status.date - self.current_date) > timedelta(seconds=1):
                    self.current_date = status.date
                    self.__send_status_message(status)
                    statuses_found += 1
            print "[" + self.myAgent.getName() + "] Found statuses: " + str(statuses_f

        def __send_registration_message(self):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("register")
            message.setContent(self.myAgent.getName())
            self.myAgent.send(message)

        def __send_status_message(self, post):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("notify")
            object = ReportMessage("Facebook", post.status_type, self.myAgent.keyword
                                   post.username, post.name, post.date, post.message +
            value = json.dumps(object.__dict__)
            message.setContent(value)
            self.myAgent.send(message)

        def __send_report_message(self):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("report")
            message.setContent(self.myAgent.getName())
            self.myAgent.send(message)

    class ReportDeliveryBehaviour(Behaviour.EventBehaviour):

        def _process(self):
            received_message = self._receive()
            if received_message:
                content = json.loads(received_message.getContent())
                print ""
                print "[" + self.myAgent.getName() + "] Received message from: " + rec
                print "[" + self.myAgent.getName() + "] Total number of fetched data:
                print ""
                for element in content:
                    notify_message = ReportMessage()
                    notify_message.load_json(element)
                    notify_message.print_message()
```

```
            self.myAgent.stop()

    def __init__(self, reporter_name, keyword, credentials_filename="credentials.json"
        agent_id, password = self.__generate_agent_credentials()
        Agent.Agent.__init__(self, agent_id, password)
        self.keyword = keyword
        self.credentials_filename = credentials_filename
        self.time = time
        self.period = period
        self.receiver = AID.aid(name=reporter_name, addresses=["xmpp://" + reporter_na

    def _setup(self):
        print "[" + self.getName() + "] Facebook fetch agent is starting..."
        fetch_behaviour = self.FetchBehaviour(self.time, self.period, self.credentials
        self.addBehaviour(fetch_behaviour)

        delivery_template = Behaviour.ACLTemplate()
        delivery_template.setOntology("report_delivery")
        self.addBehaviour(self.ReportDeliveryBehaviour(), delivery_template)

    def __generate_agent_credentials(self):
        agent_name = "facebook_" + datetime.now().strftime("%H%M%S") + "@127.0.0.1"
        return agent_name, "secret"
```

Agent ima dvije podklase koje definiraju ponašanje agenta, a to su *FetchBehaviour* i *ReportDeliveryBehaviour*.

`FetchBehaviour` je ponašanje koje je definirano kao `PeriodicBehaviour` // TODO... Sastoji se od metoda koje su definirane od strane nadklase `PeriodicBehaviour` gdje se kod pokretanja šalje poruka registracije agentu izvjestitelju o Facebook agentu pomoću metode `__send_registration_message()` kako bi agent izvjestitelj mogao voditi evidenciju o njemu i njegovim podacima. U svakom periodu provjerava se da li je broj perioda koje je korisnik definirao iskorišten. Ukoliko jest, onda se prelazi iz metode `_onTick()` na samo jedno izvršavanje metode `onEnd()` gdje se agentu izvjestitelju šalje poruka da agent želi dobiti izvješće i to pomoću metode `__send_report_message()`. U suprotnosti se šalje zahtjev za provjerom da li se na Facebook servisu što promijenilo od zadnje provjere. To je omogućeno pomoću metode `fetch_data()`. Nakon dohvaćanja podataka, ukoliko je bilo kakve promjene, dohvaćeni podaci se šalju agentu izvjestitelju pomoću metode `__send_status_message()`.

`ReportDeliveryBehaviour` je ponašanje koje je definirano kao `EventBehaviour` // TODO... Sastoji se od samo jedne metode koja je definirana nadklasom `EventBehaviour`, a radi na način da očekuje poruku od agenta izvjestitelja, a kada ju dobije ju ispiše na ekran i završi s radom agenta.

Konačno, sam agent `FacebookAgent` nasljeđuje klasu `Agent` te se time definiraju konstruktor i metoda `_setup()`. Konstruktor je u ovom slučaju definiran drugačije od zadanog, pa je potrebno ručno pozvati konstuktor i proslijediti potrebne podatke. Podaci koje je potrebno proslijediti jesu korisničko ime i lozinka agenta, a to se generira pomoću metode `__generate_agent_credentails()`.

3.2 Implementacija Twitter agenta

U ovom potpoglavlju prikazat ću način na koji sam implementirao Twitter agenta. Prije prikaza same implementacije, potrebno je naglasiti, kao i kod Facebook agenta, da je važno da se unutar datoteke *credentials.json* unesu potrebne vrijednosti za `"api_key"`, `"api_key_secret"`, `"access_token"` i `"access_token_secret"`. Navedene podatke je moguće dobiti nakon što se kreira nova aplikacija unutar Twitter aplikacije u dijelu za razvijatelje aplikacija.

3.2.1 Prikaz implementacije modela

Za dobivanje podataka potrebnih za spajanje na Twitter-ov servis, koristio sam sljedeću klasu *TwitterCredentials*. Pomoću metode `__read_configuration()` dohvaća se datoteka unutar koje se nalaze podaci o spajanju na Twitter servis, dok se pomoću metode `get_credentials()` dohvaćaju navedeni podaci iz datoteke te vraćaju kao n-torka.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

import json


class TwitterCredentials:

    def __init__(self, filename):
        self.filename = filename

    def get_credentials(self):
        json_data = self.__read_configuration_file()
        api_key = json_data["twitter"]["api_key"]
        api_key_secret = json_data["twitter"]["api_key_secret"]
        access_token = json_data["twitter"]["access_token"]
        access_token_secret = json_data["twitter"]["access_token_secret"]
        return api_key, api_key_secret, access_token, access_token_secret

    def __read_configuration_file(self):
        file = open(self.filename, "r")
        data = json.load(file)
        file.close()
        return data
```

Sljedeća klasa koja predstavlja model je *TwitterPost*. Ova klasa sadrži osnovne podatke o statusu odnosno tweet-u, a to su redom; vrijeme objave, id posljednje objave, tekst statusa, korisničko ime, ime i prezime korisnika te lokacija.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

from datetime import datetime, timedelta
```

```
class TwitterPost:

    def __init__(self, data):
        self.date = datetime.strptime(data["created_at"], "%a %b %d %H:%M:%S +0000 %Y"
        self.since_id = data["id_str"]
        self.text = data["text"]
        self.username = data["user"]["screen_name"]
        self.name = data["user"]["name"]
        self.location = data["user"]["location"]
```

3.2.2 Prikaz implementacije API-a

Sljedeće što je potrebno da bi Twitter agent radio i dohvaćao podatke je API koji definira metode pomoću kojih je moguće spajanje na Twitter servis. Na taj način se dobivaju podaci ovisno da li se radi o dohvaćanju statusa korisnika ili novih statusa na temelju hashtag-a.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

import requests
import requests_oauthlib

from social.twitter.twitter_models import *


class TwitterAPI:

    def __init__(self, credentials):
        self.__credentials = credentials

    def search_hashtag(self, hashtag, since_id=0):
        url = "https://api.twitter.com/1.1/search/tweets.json"
        params = {"q": hashtag, "result_type": "recent", "since_id": since_id}
        auth = requests_oauthlib.OAuth1(*self.__credentials)
        request = requests.get(url, params=params, auth=auth)
        hashtags = []
        for json_hashtag in reversed(request.json()["statuses"]):
            hashtag = TwitterPost(json_hashtag)
            hashtags.append(hashtag)
        return hashtags

    def search_tweets(self, username, since_id=0):
        url = "https://api.twitter.com/1.1/statuses/user_timeline.json"
        params = {"screen_name": username}
```

```
        if since_id > 0:
            params["since_id"] = since_id
        auth = requests_oauthlib.OAuth1(*self.__credentials)
        request = requests.get(url, params=params, auth=auth)
        tweets = []
        for json_tweet in reversed(request.json()):
            tweet = TwitterPost(json_tweet)
            tweets.append(tweet)
        return tweets
```

Definirane su dvije osnovne metode pomoću kojih se dohvaćaju podaci preko API-a. To su metode `search_hashtag(hashtag, since_id` i `search_tweets(username, since_id)`.

Metoda `search_hashtag(hashtag, since_id)` traži sve statuse koji sadrže navedeni hashtag, a nastali su nakon prethodnog traženja od strane korisnika. Metoda `search_tweets(username, since_id(` traži nove statuse od određenog korisnika koji su nastali nakon zadnje pretrage. Obje metode nakon pozivanja vraćaju statuse koji su dohvaćeni.

3.2.3 Prikaz implementacije reaktivnog agenta

Ovdje ću prikazati konačnu implementaciju agenta koji na Twitter društvenoj mreži dohvaća promjene u smislu novih statusa od stane korisnika ili novih statusa koji sadrže definirani hashtag.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

import json
from datetime import datetime, timedelta

from spade import Agent
from spade import Behaviour
from spade import AID
from spade import ACLMessage

from social.twitter.twitter_api import TwitterAPI
from social.twitter.twitter_credentials import TwitterCredentials
from social.shared.report_message import ReportMessage


class TwitterFetchAgent(Agent.Agent):

    class FetchBehaviour(Behaviour.PeriodicBehaviour):

        def __init__(self, time, period, credentials_filename):
            Behaviour.PeriodicBehaviour.__init__(self, period)
            self.periods = time / period
            self.current_date = datetime.now()
            twitter_credentials = TwitterCredentials(credentials_filename)
```

```
            self.twitter_api = TwitterAPI(twitter_credentials.get_credentials())

        def onStart(self):
            print "[" + self.myAgent.getName() + "] " + self.myAgent.fetch_type.capit
            self.__send_registration_message()

        def onEnd(self):
            print "[" + self.myAgent.getName() + "] " + self.myAgent.fetch_type.capit
            self.__send_report_message()

        def _onTick(self):
            if self.periods > 0:
                self.periods -= 1
                self.fetch_data()
            else:
                self.kill()

        def fetch_data(self):
            if self.myAgent.fetch_type == "tweet":
                print "[" + self.myAgent.getName() + "] Fetching tweets ... (No. of pe
                results = self.twitter_api.search_tweets(self.myAgent.keyword)
            else:
                print "[" + self.myAgent.getName() + "] Fetching hashtags ... (No. of
                results = self.twitter_api.search_hashtag(self.myAgent.keyword)

            tweets_found = 0
            for tweet in results:
                if (tweet.date - self.current_date) > timedelta(seconds=1):
                    self.current_date = tweet.date
                    self.__send_tweet_message(tweet)
                    tweets_found += 1
            print "[" + self.myAgent.getName() + "] Found " + self.myAgent.fetch_type

        def __send_registration_message(self):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("register")
            message.setContent(self.myAgent.getName())
            self.myAgent.send(message)

        def __send_tweet_message(self, tweet):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("notify")
            object = ReportMessage("Twitter", self.myAgent.fetch_type, self.myAgent.ke
```

```
                                    tweet.username, tweet.name, tweet.date, tweet.text)
            value = json.dumps(object.__dict__)
            message.setContent(value)
            self.myAgent.send(message)

        def __send_report_message(self):
            message = ACLMessage.ACLMessage()
            message.addReceiver(self.myAgent.receiver)
            message.setOntology("report")
            message.setContent(self.myAgent.getName())
            self.myAgent.send(message)

    class ReportDeliveryBehaviour(Behaviour.EventBehaviour):

        def _process(self):
            received_message = self._receive()
            if received_message:
                content = json.loads(received_message.getContent())
                print ""
                print "[" + self.myAgent.getName() + "] Received message from: " + rec
                print "[" + self.myAgent.getName() + "] Total number of fetched data:
                print ""
                for element in content:
                    notify_message = ReportMessage()
                    notify_message.load_json(element)
                    notify_message.print_message()
            self.myAgent.stop()

    def __init__(self, reporter_name, keyword, fetch_type="tweet",
                 credentials_filename="credentials.json", time=60, period=10):
        agent_id, password = self.__generate_agent_credentials()
        Agent.Agent.__init__(self, agent_id, password)
        self.keyword = keyword
        self.fetch_type = fetch_type
        self.credentials_filename = credentials_filename
        self.time = time
        self.period = period
        self.receiver = AID.aid(name=reporter_name, addresses=["xmpp://" + reporter_na

    def _setup(self):
        print "[" + self.getName() + "] Twitter fetch agent is starting..."
        fetch_behaviour = self.FetchBehaviour(self.time, self.period, self.credentials
        self.addBehaviour(fetch_behaviour)

        delivery_template = Behaviour.ACLTemplate()
```

```
        delivery_template.setOntology("report_delivery")
        self.addBehaviour(self.ReportDeliveryBehaviour(), delivery_template)

    def __generate_agent_credentials(self):
        agent_name = "twitter_" + datetime.now().strftime("%H%M%S") + "@127.0.0.1"
        return agent_name, "secret"
```

Agent ima dvije podklase koje definiraju ponašanje agenta, a to su *FetchBehaviour* i *ReportDeliveryBehaviour*.

`FetchBehaviour` je ponašanje koje je definirano kao `PeriodicBehaviour` // TODO... Sastoji se od metoda koje su definirane od strane nadklase `PeriodicBehaviour` gdje se kod pokretanja šalje poruka registracije agentu izvjestitelju o Twitter agentu pomoću metode `__send_registration_message()` kako bi agent izvjestitelj mogao voditi evidenciju o njemu i njegovim podacima. U svakom periodu provjerava se da li je broj perioda koje je korisnik definirao iskorišten. Ukoliko jest, onda se prelazi iz metode `_onTick()` na samo jedno izvršavanje metode `onEnd()` gdje se agentu izvjestitelju šalje poruka da agent želi dobiti izvješće i to pomoću metode `__send_report_message()`. U suprotnosti se šalje zahtjev za provjerom da li se na Twitter servisu što promijenilo od zadnje provjere. Ovdje je važno spomenuti da se na temelju korisničkog unosa odnosno definiranja pretrage šalje različiti zahtjev, tj. poziva se različita metoda. Metoda koja se koristi za dohvaćanje novih podataka je `fetch_data()` te ona poziva `self.twitter_api.search_tweets(username, since_id)` ako je korisnik kao vrstu pretrage odabrao *hashtag*, dok se u slučaju vrste pretrage *tweet* poziva metoda `self.twitter_api.search_hashtags(hashtag, since_id)`. Nakon što se podaci dohvate, isti se šalju agentu izvjestitelju.

`ReportDeliveryBehaviour` je ponašanje koje je definirano kao `EventBehaviour` // TODO... Sastoji se od samo jedne metode koja je definirana nadklasom `EventBehaviour`, a radi na način da očekuje poruku od agenta izvjestitelja, a kada ju dobije ju ispiše na ekran i završi s radom agenta.

Konačno, sam agent `TwitterAgent` nasljeđuje klasu `Agent` te se time definiraju konstruktor i metoda `_setup()`. Konstruktor je u ovom slučaju definiran drugačije od zadanog, pa je potrebno ručno pozvati konstuktor i proslijediti potrebne podatke. Podaci koje je potrebno proslijediti jesu korisničko ime i lozinka agenta, a to se generira pomoću metode `__generate_agent_credentails()`.

3.3 Implementacija agenta za izvještaje

Važna stavka ovog rješenja je agent izvjestitelj. Ovaj agent zadužen je za prihvaćanje svih informacija koje dobije od agenata koji dohvaćaju podatke s društvenih mreža.

Da bi agent mogao komunicirati s agentima (Facebook, Twitter), definirana je klasa koja predstavlja model poruke pomoću koje se šalju konačni rezultati odnosno izvještaj za određenog agenta. Klasa *ReportMessage* sadrži informacije o poruci kao što je vrsta mreže koja predstavlja o kojem se agentu radi (Facebook ili Twitter), vrsti poruke koja predstavlja da li se radi o statusu ili slici, traženoj riječi, korisničkom imenu, imenu i prezimenu korisnika, datumu i sadržaju statusa. Također, pošto se radi o JSON tipu poruke koja služi za komunikaciju između agenata, potrebno je moći pročitati sadržaj, a za to služi metoda `load_json(data)`. Uz to, definirana je metoda za ispis rezultata kako bi on bio definiran na razini modela, a naziv metode je `print_message()`.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-
```

```
from datetime import datetime


class ReportMessage:

    def __init__(self, network=None, message_type=None, keyword=None, username=None, n
        self.network = network
        self.message_type = message_type
        self.keyword = keyword
        self.username = username
        self.name = name
        if date:
            self.date = date.strftime("%H:%M:%S, %d.%m.%Y")
        else:
            self.date = date
        self.text = text

    def load_json(self,data):
        self.network = data["network"]
        self.message_type = data["message_type"]
        self.keyword = data["keyword"]
        self.username = data["username"]
        self.name = data["name"]
        self.date = data["date"]
        self.text = data["text"]

    def print_message(self):
        print "network=" + self.network + "; type=" + self.message_type + "; keyword="
        print "username=" + self.username + "; name=" + self.name
        print "date=" + self.date
        if self.text == "":
            print "message=<empty_message>"
        else:
            print "message=" + self.text.replace("\n", " ").replace("\r", "")
        print ""
```

Agent je implementiran u klasi *ReportAgent* te ima tri podklase koje definiraju ponašanje agenta, a to su *RegisterBehaviour*, *NotifyBehaviour* i *ReportBehaviour*.

Sva tri ponašanja definirana su kao `EventBehaviour`, a nešto više o toj vrsti ponašanja rečeno je u ranijim poglavljima.

`RegisterBehaviour` se sastoji od metode koja je definirana od strane nadklase `EventBehaviour`, a radi na način da dohvaća poruke od strane drugih agenata (Facebook, Twitter) te ih registrira na način da ih spremi u adresar (engl. dictionary). Pošto svaki od agenata ima drugačije ime, adresar je dobar način spremanja pošto ime kao ključ nikad neće biti isto.

`NotifyBehaviour` se sastoji od metode koja je definirana od strane nadklase `EventBehaviour`, a radi na način da dohvaća poruke od strane drugih agenata (Facebook, Twitter). Ukoliko je poruka primljena, poruka se otpakira iz JSON formata u određeni model te se kao takva sprema u rječnik i ispisuje.

`ReportBehaviour` se sastoji od metode koja je definirana od strane nadklase `EventBehaviour`, a radi na način da dohvaća poruke od strane drugih agenata (Facebook, Twitter). Navedeni agenti kada su pri kraju s radom šalju zahtjev za konačnim izvješćem od strane agenta izvjestitelja, pa agent dohvaća iz rječnika i šalje navedenom agentu. Nakon što se poslala poruka s podacima, isti se agent briše iz rječnika. Metoda pomoću koje se šalje poruka je `send_report(agent_name)`.

Agent `ReportAgent` nasljeđuje klasu `Agent` te se time definiraju konstruktor i metoda `_setup()`. Unutar ove metode potrebno je definirati predloške koji će biti primjenjeni za svako od navedenih ponašanja kako bi poruke stizale samo za navedena ponašanja.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

import json

from spade import Agent
from spade import Behaviour
from spade import ACLMessage
from spade import AID

from social.shared import report_message


class ReportAgent(Agent.Agent):

    class RegisterBehaviour(Behaviour.EventBehaviour):

        def _process(self):
            received_message = self._receive()
            if received_message:
                agent_name = received_message.getContent()
                self.myAgent.reports[agent_name] = []
                print "[" + self.myAgent.getName() + "] New agent (" + agent_name + "
            print "[" + self.myAgent.getName() + "] No. of agents working: " + str(len

    class NotifyBehaviour(Behaviour.EventBehaviour):

        def _process(self):
            received_message = self._receive()
            if received_message:
                content = received_message.getContent()
                notify_message = report_message.ReportMessage()
                notify_message.load_json(json.loads(content))
```

```
            self.myAgent.reports[received_message.getSender().getName()].append(n
            print "[" + self.myAgent.getName() + "] Received message from: " + rec
            notify_message.print_message()

class ReportBehaviour(Behaviour.EventBehaviour):

    def _process(self):
        received_message = self._receive()
        if received_message:
            agent_name = received_message.getContent()
            if agent_name in self.myAgent.reports:
                self.send_report(agent_name)
                del(self.myAgent.reports[agent_name])
                print  "[" + self.myAgent.getName() + "] Agent (" + agent_name + "
        print "[" + self.myAgent.getName() + "] No. of agents working: " + str(len

    def send_report(self, agent_name):
        report_messages = self.myAgent.reports[agent_name]
        receiver = AID.aid(name=agent_name, addresses=["xmpp://" + agent_name])
        message = ACLMessage.ACLMessage()
        message.addReceiver(receiver)
        message.setOntology("report_delivery")
        message.setContent(json.dumps([obj.__dict__ for obj in report_messages]))
        self.myAgent.send(message)

def _setup(self):
    print "[" + self.getName() + "] Report agent is online."
    self.reports = {}

    register_template = Behaviour.ACLTemplate()
    register_template.setOntology("register")
    self.addBehaviour(self.RegisterBehaviour(), register_template)

    notify_template = Behaviour.ACLTemplate()
    notify_template.setOntology("notify")
    self.addBehaviour(self.NotifyBehaviour(), notify_template)

    report_template = Behaviour.ACLTemplate()
    report_template.setOntology("report")
    self.addBehaviour(self.ReportBehaviour(), report_template)
```

3.4 Implementacija kreiranja agenata

Na kraju, vrijedi prikazati implementaciju klase *SocialNotifier* unutar koje je definirano na koji način se pokreće program. Korištena je biblioteka *argparse* koja omogućuje definiranje korištenja argumenata na vrlo jednostavan način, odnosno da se definira broj potrebnih argumenata, koliko je obaveznih te koliko opcionalnih argumenata i na kraju da se navedeni podaci vrlo lako koriste u daljnjem izvođenju programa. U programskom kodu je vidljivo da se definira `parser` koji obuhvaća tri načina unosa i to kao `reporter_parser`, `twitter_parser` i `facebook_parser`. U ovisnosti o načinu pokretanja programa, pokrenut će se drugačiji agent i to:

- ReportAgent
- FacebookAgent
- TwitterAgent

Nakon provjere argumenata, provjerava se koja je od komandi unesena što predstavlja zapravo naziv prvog argumenta. Ukoliko je komanda unosa odgovarajuća, poziva se jedna od navedenih metoda ovisno u nazivu komande; `__start_reporter(result)` za pokretanje *ReportAgent*-a, `__start_twitter_fetch(result)` za pokretanje *TwitterAgent*-a te `__start_facebook_fetch(result)` za pokretanje *FacebookAgent*-a.

Posljednja metoda koja se nalazi u kodu jest `__read_config_file(filename)` pomoću koje se dohvača sadržaj datoteke *config.json* u kojoj se nalazi korisničko ime i lozinka agenta izvjestitelja.

```
#! /usr/bin/usr python
# -*- coding: utf-8 -*-

import argparse
import sys
import json
from os import path

from agents.facebook_agent import FacebookAgent
from agents.twitter_agent import TwitterFetchAgent
from agents.report_agent import ReportAgent


class SocialNotifier:

    def __init__(self):
        self.parse_parameters()

    def parse_parameters(self):
        parser = argparse.ArgumentParser()
        subparsers = parser.add_subparsers(dest="command")

        # Reporter
        reporter_parser = subparsers.add_parser("reporter", help="Creates social repo
        reporter_parser.add_argument("-config", default="config.json", action="store"
```

```
        # Twitter
        twitter_parser = subparsers.add_parser("twitter", help="Creates a Twitter fetc
        twitter_parser.add_argument("keyword", action="store", help="Defines search ke
        twitter_parser.add_argument("type", action="store", help="Defines search type.
        twitter_parser.add_argument("-credentials", default="credentials.json", action
        twitter_parser.add_argument("-config", default="config.json", action="store",
        twitter_parser.add_argument("-time", default=120, action="store", help="Length
        twitter_parser.add_argument("-period", default=15, action="store", help="Lengh

        # Facebook
        facebook_parser = subparsers.add_parser("facebook", help="Creates a Facebook f
        facebook_parser.add_argument("keyword", action="store", help="Defines search k
        facebook_parser.add_argument("-credentials", default="credentials.json", actio
        facebook_parser.add_argument("-config", default="config.json", action="store"
        facebook_parser.add_argument("-time", default=120, help="Length of agent lifet
        facebook_parser.add_argument("-period", default=15, help="Lenght of period in

        result = parser.parse_args()
        if not path.isfile(result.config):
            sys.exit("Error: No configuration file found.")

        if result.command == "reporter":
            self.__start_reporter(result)
        elif result.command == "twitter":
            if not path.isfile(result.credentials):
                sys.exit("Error: No credentials file found.")
            if result.type != "tweet" and result.type != "hashtag":
                sys.exit("Error: Type must be 'tweet' or 'hashtag'.")
            self.__start_twitter_fetch(result)
        elif result.command == "facebook":
            if not path.isfile(result.credentials):
                sys.exit("Error: No credentials file found.")
            self.__start_facebook_fetch(result)
        else:
            sys.exit("Error: Problem with parameters.")

    def __start_reporter(self, result):
        name, password = self.__read_config_file(result.config)
        reporter = ReportAgent(name, password)
        reporter.start()

    def __start_twitter_fetch(self, result):
        reporter_name, _ = self.__read_config_file(result.config)
        agent = TwitterFetchAgent(reporter_name, result.keyword, result.type, result.
```

```
        agent.start()

    def __start_facebook_fetch(self, result):
        reporter_name, _ = self.__read_config_file(result.config)
        agent = FacebookAgent(reporter_name, result.keyword, result.time, result.perio
        agent.start()

    def __read_config_file(self, filename):
        file = open(filename, "r")
        json_data = json.load(file)
        agent_name = json_data["agent_name"]
        agent_password = json_data["agent_password"]
        file.close()
        return agent_name, agent_password


if __name__ == "__main__":
    social_notifier = SocialNotifier()
```

Poglavlje 4

Demonstracija rješenja

Poglavlje 5

Zaključak

Bibliografija

..., 2004. Deklaracija o znanju - Hrvatska temeljena na znanju i primjeni znanja. Hrvatska akademija znanosti i umjetnosti, Zagreb, Croatia.

Abele, T. and Bischoff, V., 2001. *Fraktal+: Adaptability in the Age of E-Business and Networking,*. In Innovations for an e-Society. 1–6.

Bahr, A., 2009. Cooperative Localization for Autonomous Underwater Vehicles. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Baral, C., 2004. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge, New York, Port Melbourne, Cape Town.

Bača, M., Schatten, M. and Deranja, D., 2007. *Autopoietic Information Systems in Modern Organizations*. Organizacija, Journal of Management, Informatics and Human Resources, Vol. 40, 3, 157–165.

Bača, M., Schatten, M. and Rabuzin, K., 2006. *A Framework for Systematization and Categorization of Biometrics Methods*. In M. Bača and B. Aurer (eds.) International Conference on Information and Intelligent Systems – IIS2006 Conference Proceedings. Faculty of Organization and Informatics, 271–278.

Berger, S., 2006. *Pythologic – Prolog syntax in Python*. Available at `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/303057`.

Garzarelli, G., 2004. *Open Source Software and the Economics of Organization*. In J.G. Birner (ed.) Markets, Information and Communication, Routledge, London and New York. 47–62.

Jennex, M.E., 2007. Knowledge Management in Modern Organizations. Idea Group Publishing, Hershey, London, Melbourne, Singapore.

Johansen, R. and Swigart, R., 2000. Upsizing The Individual In The Downsized Corporation Managing In The Wake Of Reengineering, Globalization, And Overwhelming Technological Change. Perseus Publishing.

Jurin, E., 2006. *Blogosfera u novoj komunikacijskoj areni*. manager.hr, Poslovni Svijet, Vol. XVII, 1025, 22.

Luhmann, N., 2003. *Organization*. In T. Bakken and T. Hernes (eds.) Autopoietic Organization Theory Drawing on Niklas Luhmann's Social Systems Perspective, Abstract, Liber, Copenhagen Business School Press, Oslo. 31–53.

Pilgrim, M. *Dive into Python*. Available at `http://diveintopython.org/`.

Pogačnik, M. and Bloom, W., 1998. Zmajeve linije – energetske mreže zemlje. Quantum, Zagreb, Croatia.

Schatten, M., 2008. Zasnivanje otvorene ontologije odabranih segmenata biometrijske znanosti. M.sc. diss., Faculty of Organization and Informatics, Varaždin.

van der Blonk, H., Huysman, M. and Spoor, E., 1998. *Autopoiesis and the evolution of information systems*. Tech. rep.