Maciej KORYL
Politechnika Rzeszowska

RESOURCES-BASED CONCEPT OF COMPUTATION FOR ENTERPRISE SOFTWARE

Summary. Traditional computational models for enterprise software are still to a great extent centralised. However, rapid growing of modern computation techniques and frameworks causes that contemporary software becomes more and more distributed. Towards development of new complete and coherent solution for distributed enterprise software construction, synthesis of three well-grounded concepts is proposed: Domain-Driven Design technique of software engineering, REST architectural style and actor model of computation. As a result new resources-based framework arises, which after first cases of use seems to be useful and worthy of further research.

KONCEPCJA PRZETWARZANIA DLA OPROGRAMOWANIA KORPORACYJNEGO BAZUJĄCA NA ZASOBACH

Streszczenie. Tradycyjne modele przetwarzania dla oprogramowania korporacyjnego są modelami scentralizowanymi, jednak szybki wzrost nowoczesnych technik obliczeniowych oraz dostępnych bibliotek i frameworków powoduje, że współczesne oprogramowanie staje się coraz bardziej oprogramowaniem rozproszonym. Zdążając w kierunku opracowania nowego całościowego i spójnego rozwiązania dla rozproszonych systemów korporacyjnych, w niniejszej pracy zaproponowano syntezę trzech dobrze ugruntowanych koncepcji: podejścia Domain-Driven Design, stylu architektonicznego REST i modelu aktorowego. W rezultacie otrzymano użyteczne rozwiązanie, w którym przetwarzanie rozproszone odbywa się w oparciu o zasoby.

1. Introduction

Enterprise software systems working in deployment environment of huge corporations such as banks or industrial plants consist of many separate products, typically from several to several dozen parts. One software product serves from hundreds to above thousand use cases and at the same time interacts with use cases of other products due to automation of complex business processes. From automation kind point of view, two types of processes may be listed:

- processes of interactive character, often automated by usage of workflow tools, responsible for entire process composition by using atom elements representing well defined and cohesive activities. Process composition in such manner is called orchestration;
- processes of batch character, consisted of processing fragments one following another or one running parallel with another, typically iterated on collections of business objects characteristic for particular area, such as contracts, transactions, orders and so. Nowadays implementations of such batch processes are supported by modern software frameworks dedicated to that purpose.

In both cases, constructed processes consist of many functions working in close cooperation, often exposed as APIs of systems or APIs of systems' components. As long as cooperation is carried in synchronous way, complexity of such arrangement may be controlled, even if number of involved systems is substantial and number of interactions is high. In synchronous systems number of possible states in which system may be is countable and predictable on the stage of software designing or detectable during testing. After applying asynchronous model of communication, complexity of system violently grows with increase of possible different states, which number is non-linear function of possible states of constituents of the system and number of interactions between them ([4]). During many years of computation theories development and many years of software engineering practices implementation, meaningful conceptual and technical tools were established, but that does not mean that problem of complexity has passed away or even has been minimized to notable degree. The matter is broadly recognized in specialized computation areas dealing with well-established algorithms, but still is not enough captured in commercial products' development, govern by its own specificity connected with high number of software users, many different business objects and huge number of unpredictable interactions (good characteristic of such systems is shown in [8]). In these days, problem is more and more complicated because of limitations of monolithic systems and need for introduction of distributed software, for example in the form of microservices ([12]), which are adapted for horizontal scaling and well suited in actual hardware capabilities. Several conceptual and technical tools currently used in software engineering discipline, dedicated to distributed processing are described in [3], where one of them is an actor model of computation, which after many years of academic development, currently gains great popularity in commercial area.

Proposition which is shown in this paper constitutes coherent and complete framework based on well-tried techniques and design patterns with actor model between them and has working implementation in Java programming language with use of modern tools for enterprise applications such as Spring Framework and noSQL databases. Proposed solution has been used to build some parts of banking transactional system, which supports batch processes such as massive transactions processing or financial instruments valuation. As a foundation of the idea three engineering concepts act: Domain-Driven Design technique proposed in [5] and broadly accepted in software community, Representational State Transfer architectural style introduced in [6] and currently becoming the most popular way of interaction between web components, and the actor model of computation described in [10] nowadays gaining mature and useful implementations. In addition, the solution was enriched by use of standard language of agent communication in multi-agent systems – Agent Communication Language, which semantics was found as very suitable for required interactions.

Plan for this paper is as follows: on the beginning base concepts used are shortly described, next central idea of this paper - the emergent resources concept is explained, after that example of interaction in real banking system supported by new framework is shown, and finally closing remarks and plans for further work appear.

2. Fundamental concepts

Resource as a central point of the REST model. Representational State Transfer (REST) is an architectural style implementing fundamental rules of the web and HTTP standard. REST was introduced by dissertation [6] and at present has obtained great popularity as "web used correctly". Central idea of the style is to treat all things which have identity as resources and give them globally unique Uniform Resource Identifier (URI). Resources named in this way may communicate together using hyperlinks. Communication is provided by usage of standard HTTP commands with their established semantics. Important rule of REST is that communication ought to be stateless, i.e. parties cannot keep state of communication assuming that next message will be continuation of previous one. In proposed approach the resource term plays key role as external representation of computation units and set of REST rules and good practices in interactions modelling are applied.

Aggregate in Domain-Driven Design concept. Domain-Driven Design (DDD) concept introduced in [5] is an approach to software development which pays attention to key meaning of domain model in software design. Domain model plays central role in whole process of development acting as universal medium of communication between all participants and providing stable base for software structure. DDD technique is divided into two stacks of patterns: strategic and tactical ones. First of them serves as toolset for taking control over complexity of extensive software and second consists of a set of building blocks, which is sufficient for complete design of each kind of enterprise software on some level of abstraction. The most important pattern from tactical stack is the aggregate building block and there is plenty of rules explained in literature, how to build useful aggregates (for example [18]). Aggregate is a graph of objects tied together into one coherent object offering common set of services for external world. The only way to access aggregate constituents' capabilities is aggregate root, the central entry point to that software unit. Thanks to such construction, aggregate guarantees the consistency of changes of whole structure, controls its internal state and gives convenient way for its access. In proposed framework, aggregate plays important role as representation of stable state of a resource and also as a part of resource in dynamical state by offering its behavioural capabilities.

Actor model of computation. The actor model of computation developed many years ago and firstly published in [10] was thought as conceptual tool for understanding of concurrency. Many software frameworks based on actor model have been built to this day, but broad utilization in enterprise software area is still scarce. Currently, attention in that idea is growing, stimulated by development of multicore processors and development of cloud computing solutions with necessity of computation distribution. Theory of actor model treats actors as universal primitives with a capability to carry out each kind of needed computation ([9]). Actors are independent units of computation loosely coupled together, only by asynchronous message passing and the only knowledge which actor has about other actor is its mailbox address. When an actor receives a message it may do some computation, send messages to participants, create additional actors as its children or may change its own behaviour preparing itself for future course of situation. In proposed framework, actors will support implementation of dynamical state of resource.

Agent Communication Language. Agent Communication Language (ACL) is a definition of standard language used in multi-agent systems to model conversations between involved parties. Its origins are in philosophical theory of speech acts ([15]), which state that each utterance has not only informative, but also performative function, i.e. carries with itself consequences in receiver's activity. ACL has been drawn up by Foundation for Intelligent Physical Agents (FIPA) as FIPA-ACL set of standards [7]. In proposed solution ACL syntactics and semantics are used to model communication between resources, especially by use of standard vocabulary of performatives denoting the type of communicative acts.

3. Emergent resources model of computation

The emergent resources term. Term "emergent" is adapted from theory of emergent properties [13], but now is applied only in very narrow meaning to denote object which properties are in continuous change and observer cannot have knowledge about its state, but can have only some beliefs. However, future work is planned on broader exploration of that theory assuming its usefulness in explanation and construction of modern software artefacts. As "emergent resources" are considered resources, which at moment of interaction may be under change originated from other interaction, computation processes or any other factor. As emergent resource is in unstable state and its properties may change in time, another object cannot assume that something is true about that resource, even if resource still exists or not. For example, if some procedure in banking system completes payment and has information about sufficient balance of debited account, it cannot assume that payment will be successful. It ought to be ready for receiving information from target resource that operation has succeeded or not.

From software design point of view, emergent resource is represented by synthesis of three concepts:

- REST resource, which brings unambiguous global identification of resource and convenient language of communication for presentation and change of its state. It also provides availability of many technical frameworks ready for use for implementation of interactions;
- DDD aggregate, which gives a comprehensive way of modelling software external and internal structure, its behaviour and rules forming objects identity. DDD technique also brings possibility of effective and cheap implementation thanks to help of modern frameworks for enterprise software such as Spring Framework [16] which was broadly used to implement proposed solution;

- actor, which offers its capability of long-term existence and sophisticated communication abilities. Utilization of actor model is possible and reliable due to existence of mature implementations such as Akka Framework [1], which was used with support of patterns based on it ([2, 19, 20]).

Two areas of resources. Any resource in the solution may stay in one of two states: stable state, when no change of its properties is possible and emergent state, when its properties may dynamically change. Therefore, symbolically two areas are distinguished: stable resources area and emergent resource area as was presented on Fig. 1.

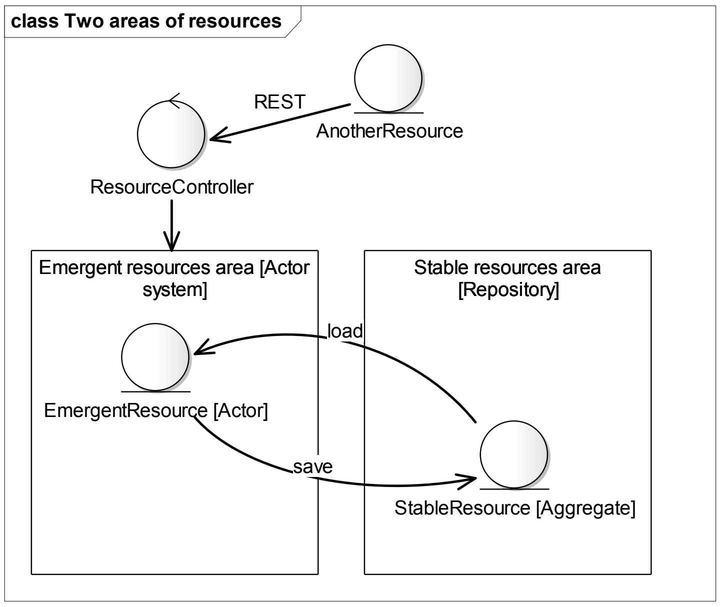
class Two areas of resources
REST
AnotherResource
ResourceController
Emergent resources area [Actor system]
Stable resources area [Repository]
load
EmergentResource [Actor]
save
StableResource [Aggregate]

Fig. 1. Two areas of resources

If message to resource in stable area was directed, resource is moved to emergent area, where it acquires ability to act. If system detects that emergent resource is idle (does not perform any activity and has empty mailbox), resource may be removed from emergent area, but it depends on strategy of supervising in use. In the system implementation these areas as represented by DDD repository pattern and by actor system respectively. For resource migration into emergent area, dedicated to such kind of resources, area supervisor is responsible. Sample body of supervisor's callback function of message handling is shown below:

```
public void onReceive(Object message) {

  if (message instanceof CreateResource) {
    // request to create new resource
    CreateResource msg = (CreateResource) message;

    // create emergent resource
    ActorRef emergentResource = context()
        .actorOf(componentName(),
            msg.getResourceId());

    // and send message to the newborn in emergent
    // state. It will be responsible for immediate
    // creation of its stable representation
    emergentResource.tell(msg, self());
  } else if (message instanceof PerformAct) {
    // message to existing resource
    PerformAct msg = (PerformAct) message;

    // is resource in emergent state?
    ActorRef emergentResource = this.getContext()
        .getChild(msg.getResourceId());

    if (emergentResource == null) { // no
    // enter existing resource into emergent state
      emergentResource = context()
          .actorOf(componentName(),
              msg.getResourceId());
    }

    // send message to resource in emergent state
    emergentResource.tell(msg, self());
  }
}
```

4. Sample interaction supported by the new model

Each processing routine in transactional system may be treated as emergent resource. Examples of such resources are: standing orders processing, interest calculation, interest capitalization, incoming or outgoing payments processing etc. Initialization of processing is therefore implemented as request for creation of resource of some kind. System ordering computation sends request to microservice which is responsible for handling such processing. Typically it will be microservice which owns resources being processed, for example customer contracts or registered payments. It also may be separate microservice as in example below, when processing involves different resources. Ordering system sends POST command with REQUEST performative and in return receives URI of emergent resource which probably will be created in the target system. Ordering system have to be ready to accept confirmation of resource creation sent by target system – the CONFIRM performative meaning that processing has been started. Thanks to received URI, ordering system is able to contact with emergent resource in target system and ask it about its state (QUERY_REF performative sent by GET command) or to order further requests, for example hold computation or abandon it (CANCEL performative sent by POST command). Ordering system should be ready to accept information from resource (which may be sent as INFORM performative by POST command) or explicitly order such information (e.g. results of computation). Sequence diagram presented on Fig. 2 illustrates example of interactions between micoservices working on some kind of processing in banking software. For clarity only single set of interactions for one transaction was shown.

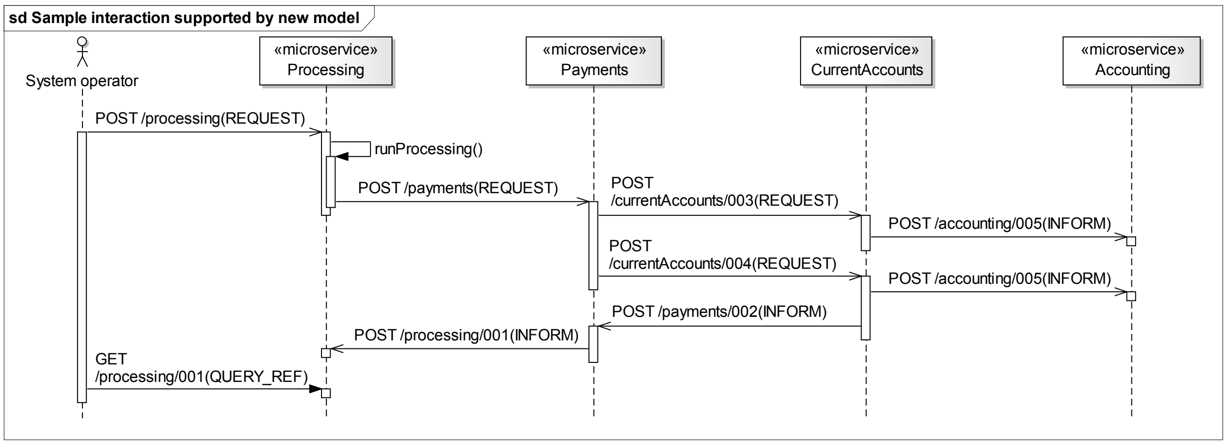
sd Sample interaction supported by new model
System operator
«microservice» Processing
«microservice» Payments
«microservice» CurrentAccounts
«microservice» Accounting
POST /processing(REQUEST)
runProcessing()
POST /payments(REQUEST)
POST /currentAccounts/003(REQUEST)
POST /accounting/005(INFORM)
POST /currentAccounts/004(REQUEST)
POST /accounting/005(INFORM)
POST /payments/002(INFORM)
POST /processing/001(INFORM)
GET /processing/001(QUERY_REF)

Fig. 2. Sample interaction supported by the new model

5. Closing remarks

New concept of computation in enterprise software and one example of its application in some processing routine in transactional system were presented. Similar solution works in real banking software and currently is under further research and development, but first results are very promising. After a few first cases of use of new solution, there might be told that characteristic features of development process based upon new framework are low cost of use cases implementation and low level of defects detected during quality assurance phase. The concept of emergent resource is thought as a broader idea and may serve as fundamental framework able to maintain whole transactional solution. Especially it probably may be applied to serve processes of interactive character, which are more unpredictable than batch processes. Such research is currently going on. Next planned stage is attempt to build complete new module of transactional system with dominance of resources of new kind. Additional direction of theoretical and practical development will be step towards utilization of some more sophisticated ideas, such as emergent properties theory, speech acts theory and indeterminism with hope, that they may help in better understanding of complex software processes.

REFERENCES

1. Akka Framework, http://akka.io.
2. Allen J.: Effective Akka. O'Reilly Media, Inc., 2013.
3. Butcher P.: Seven Concurrency Models in Seven Weeks. Pragmatic Bookshelf, 2014.
4. Chandy K. M., Lamport L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985, p. 63–75.
5. Evans E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
6. Fielding R. T.: Architectural Styles and the Design of Network-based Software Architectures. PhD dissertation, University Of California, Irvine, 2000.
7. FIPA Standards, http://www.fipa.org/repository.

8. Fowler M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
9. Hewitt C.: Actor Model of Computation: Scalable Robust Information Systems. Cornell University Library, arXiv:1008.1459, 2015.
10. Hewitt C., Bishop P., Steiger R.: A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI'73 Proceedings of the 3rd International Joint Conference on Artificial Intelligence, 1973, p. 235–245.
11. Millett S., Tune N.: Patterns, Principles, and Practices of Domain-Driven Design. John Wiley & Sons, 2015.
12. Newman S.: Building Microservices. Designing Fine-Grained Systems. O'Reilly Media, 2015.
13. O'Connor T.: Emergent Properties. American Philosophical Quarterly, 31, 1994, pp. 91-104.
14. Nash M., Waldron W.: Applied Akka Patterns. O'Reilly Media, Inc., 2016.
15. Searle J. R.: Speech Acts. Cambridge University Press, 1969.
16. Spring Framework, https://spring.io.
17. Sukumar G.: Distributed Systems: An Algorithmic Approach. Chapman and Hall/CRC, 2014.
18. Vernon V.: Implementing Domain-Driven Design. Addison-Wesley Professional, 2013.
19. Vernon V.: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015.
20. Wyatt D.: Akka Concurrency. Artima Press, 2013.