ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

PROGRESSIVE INFORMATION TECHNOLOGIES

UDC 614.2+574/578+004.38

CRITICAL CAUSAL EVENTS IN SYSTEMS BASED ON CQRS WITH EVENT SOURCING ARCHITECTURE

Lytvynov O. A. – PhD, Associate Professor of the Department of Electronic Computing Machinery, Oles Honchar Dnipro National University, Dnipro, Ukraine.

Hruzin D. L. – Postgraduate student of the Department of Electronic Computing Machinery, Oles Honchar Dnipro National University, Dnipro, Ukraine.

ABSTRACT

Context. The article addresses the problem of causal events asynchrony which appears in the service-oriented information systems that does not guarantee that the events will be delivered in the order they were published. It may cause intermittent faults occurring at intervals, usually irregular, in a system that functions normally at other times.

Objective. The goal of the work is the comparison and assessment of several existing approaches and providing a new approach for solving the causal events synchronization issue in application to the systems developed using Command Query Responsibility Segregation (CQRS) with Event Sourcing (ES) architecture approach.

Methods. Firstly, the method of estimation of the likelihood of causal events occurring within the systems as the foundation for choosing the solution is suggested. Based on the results of the analysis of several projects based on CQRS with ES architecture it shows that the likelihood of critical causal events depends on the relationships among entities and the use-cases connected with the entities. Secondly, the Container of Events method, which represents a variation of event with full causality history, adapted to the needs of CQRS with ES architecture systems, was proposed in this work. The variants of its practical implementation have also been discussed. Also, the different solutions, such as Synchronous Event Queues and variation of Causal Barrier method were formalized and assessed. Thirdly, the methods described have been discussed and evaluated using performance and modification complexity criteria. To make the complexity-performance comparative assessment more descriptive the integrated assessment formula was also proposed.

Results. The evaluation results show that the most effective solution of the issue is to use the Container of Events method. To implement the solution, it is proposed to make the modifications of the Event Delivery Subsystem and event handling infrastructure.

Conclusions. The work is focused on the solution of the critical causal events issue for the systems based on CQRS with ES architecture. The method of estimation of the likelihood of critical causal events has been provided and different solutions of the problem have been formalized and evaluated. The most effective solution based on Container of Events method was suggested.

KEYWORDS: Service-Oriented Architecture, Event-Driven Architecture, Event Sourcing, Events synchronization, Domain Driven Design.

ABBREVIATIONS

CQRS is a Command Query Responsibility Segregation;

DDD is a Domain Driven Design;

DL is a Description logics;

EDS is an event-delivery subsystem;

ES is an Event Sourcing;

HSSM is a Halstead Software Science Metrics;

IoT is an Internet of Things;

SQL is a Structured Query Language.

NOMENCLATURE

α is a weight of integration complexity in comparison with maintenance complexity;

β is a weight of maintenance complexity in comparison with integration complexity;

Δ is a bounded lifetime of a broadcast message;

ε is a set of all events occurred in domain;

$\mathcal{N}$ is a set of all notifications – messages sent by all the publishers about the events;

ρ is a weight of performance in comparison with complexity;

τ_j is an interval of subscription;

a, b, c are events;

A is a certain type of events;

$\mathfrak{C}$ is a set of event types related to a subset of events;

C_i is a complexity of method's integration;

C_i is a clock function be Lamport;

C_m is a complexity of maintenance the system with integrated method;

$C(a)$ is a timestamp of the event *a*;

$C(A)$, $C(B)$, $C(H)$, $C(P)$, $C(S)$ are use cases connected with creation;

e_a^c, e_b^c are causal events connected with the creation of the instances;

DOI 10.15588/1607-3274-2024-3-11

e_a^r, e_b^r are events connected with the removing of the instances;

e_a^m, e_b^m are events connected with the modification of the instances;

e_1, e_2, e_k are 1-st, 2-nd, *k*-th events;

E is a set of events of a distributed computation;

$E_{a+b+c}^{U_i}, E_{a+b}^{U_i}, E_{a+c}^{U_i}, E_a^{U_i}, E_b^{U_j}, E_c^{U_k}$ are event types for specific use case U_i

E_1, E_n are events of a distributed computation;

E_i is a subset of events of a certain type;

E_{imt} is an integrated performance-complexity metric;

$h_{2+}, h_{2-}, h_{2.1}$ is a part of handler responsible for processing events $< e_1, e_2 >$ in Causal Barrier variant;

ex_2 is an exceptional situation when e_2 is lost or can be considered as lost after a defined period of time, i.e. bounded lifetime Δ has expired;

H_c is a set of command handlers;

H_e is a set of event handlers;

H(*b*) is a causal history of the event *b*;

H(*Ej*) is a set of handlers responsible for processing different combinations of events from the *Ej* group of events;

I_j is a subset of incoming events which *j*-th event handler r_j, is subscribed to;

k is a number of causal events within the *Ej* group;

l is an order of magnitude;

m is a modification function which denotes the applying transformations to the existing system;

M(*A*), *M*(*B*) are use cases connected with modification;

n is a number of connected events within the *Ej* group;

n_ℓ, n_u, m_ℓ, m_u are multiplicity coefficients;

n_k is a *k*-th notification;

n_1 is a represents the count of distinct operators;

n_2 is a represents the count of distinct operands;

N_1 is a total number of operators;

N_2 is a total number of operands;

ntf() is a notify function;

O is a set of four basic interface operations;

P_{avg} is an average relative performance;

P_{hl} is an average high load performance;

P_{hpl} is an average high parallel load performance;

P_k is a relative performance of the *k*-th method;

P_i is a subset of the events published by w_i;

P_{ll} is an average low load performance;

P_y is a set of all notifications published by the command handler;

pub() is a publish function;

r is a remove predicate;

r_j is a *j*-th event handler;

R(*A*), *R*(*B*) are use cases connected with removing;

$s, t, u, v, u', v', u'', v''$ are time markers;

s_A is a subscription to *A*-type events;

S_x is a set of active subscriptions for event handler;

sub() is a subscribe function;

T_k is a represents the time metric;

$T_{\min}$ is a represents the lowest time metric across all compared metrics;

$U_i(A)$, $U_i(B)$ are use case;

usub() is a unsubscribe function;

w_i is a *i*-th command handler;

w, w_1, w_2 are worlds, using Kripke semantics;

W is a set of worlds, using Kripke semantics;

□ is a necessary truth;

◊ is a possibility;

¬◊ is a impossibility.

INTRODUCTION

Development of the Modern software is an essential part of any business which helps to increase productivity, reduce costs, and improve customer services. But the modern business is always under the influence of changing business rules, adding new activities, modifications of procedures and processes. Thus, the systems developed as a business infrastructure should be flexible enough to be adapted to business and system requirements changes as quickly as possible. To handle this challenge different approaches [1], principles [2] and architectures [3–6] are provided.

One of the effective solutions is to build the system using event-driven architecture [4] which is based on Publisher-Subscriber pattern [7] of communication. It allows to enable indirect communication between modules (usually cloud services [5, 6]) using an intermediate infrastructure called Event Publisher which is responsible for delivering the messages published by the publishers to the subscribers. And thus, it allows to increase the level of flexibility [8] and scalability of the system.

Whilst event-driven and service-oriented architectures offer advantages at the system level, the combination of the Command Query Responsibility Segregation (CQRS) with Event Sourcing (ES) architectural design patterns is frequently employed in such systems to enhance application-level performance [9–11].

CQRS [12] is a design pattern that separates the command (write) side of an application from the query (read) side. CQRS is used in conjunction with ES [13] to provide a clear separation between the handling of commands that change the application state and the retrieval of data for querying. During a write operation, events are recorded in the event store, and the client is informed that the source of truth of the system has been updated, and eventually [14], the other parts of the system (e.g. projections [15], services) will be updated. The projections, which are denormalized data representations stored in the format requested by the client, are eventually updated by the event handlers subscribed to certain events. Projections may be based on SQL or NoSQL databases, or even pre-rendered web pages.

DOI 10.15588/1607-3274-2024-3-11

The main advantages of CQRS with ES architecture are as follows. Write operations are performed quickly in comparison to the non-CQRS systems, because the execution of the commands does not depend on database manipulation, and data manipulation is restricted only to saving events, which appear in the result of the command execution, to the Event Store. Read operations are reduced to selecting pre-prepared data causing significant speed-up of user query processing. Clear separation of concerns between commands and queries, facilitating a more modular and maintainable codebase.

The CQRS with ES architecture is the most applicable for systems that are based on events on a business level, e.g. trip systems, financial systems [16]. But this architecture is not applicable to systems that require a strong degree of temporal consistency [17].

The systems based on CQRS with ES architecture can also use the Publisher-Subscriber pattern to realize flexible, indirect communication between command handlers, which are the producers and publishers of the events, to the event handlers, which are the subscribers of the events.

The object of the study is a causal events phenomenon which appears in information systems.

One of the important problems which appears in the systems based on CQRS with ES architecture which uses Publisher-Subscriber pattern for enabling indirect communication between command handlers and events handlers is the problem of synchronization of causal events. Causal events are causally related events the order of which should be preserved, i.e. the events connected by happened-before relationship [18, 19]. The source of the problem is the fact that the event delivery subsystem does not guarantee that published events will be delivered in the order they were published. It may cause intermittent, hardly detected faults occurring at intervals, usually irregular, in a system that functions normally at other times. The existing solutions of this problem are not formalized and evaluated, the likelihood of appearance of the issue is not well understood. Therefore, to provide an effective solution to this problem, preserving the maintainability level of the system, it is necessary to study and evaluate the existing methods of the solution.

The subject of the study is the issue of synchronization of causal events in systems based on CQRS with ES architecture.

The purpose of the work is evaluating the likelihood of the appearance of causal events in systems based on CQRS with ES architecture and provide the most effective method of the solution to the problem based on the results of the evaluation of different existing and novel solutions using complexity and performance criteria.

1 PROBLEM STATEMENT

Among several well-known problems connected to CQRS with ES architecture [20, 21] the problem of critical causal events synchronization has not been fully studied. Perhaps, the reason is that using Publisher-Subscriber pattern is not only one way to realize the communication between Write and Read subsystems. The source of the problem is connected to Publisher-Subscriber pattern and inability of the event delivery subsystem to guarantee the preservation of the order of published events, i.e. that published events will be delivered in the order they were published. It may cause intermittent synchronization faults which are considered as one of the most difficult problems in distributed programming [22].

The main phases of the typical workflow of the command processing by the system built using CQRS with ES architecture and Publisher-Subscriber pattern-based communication subsystem considered in this paper are as follows (See Fig. 1).

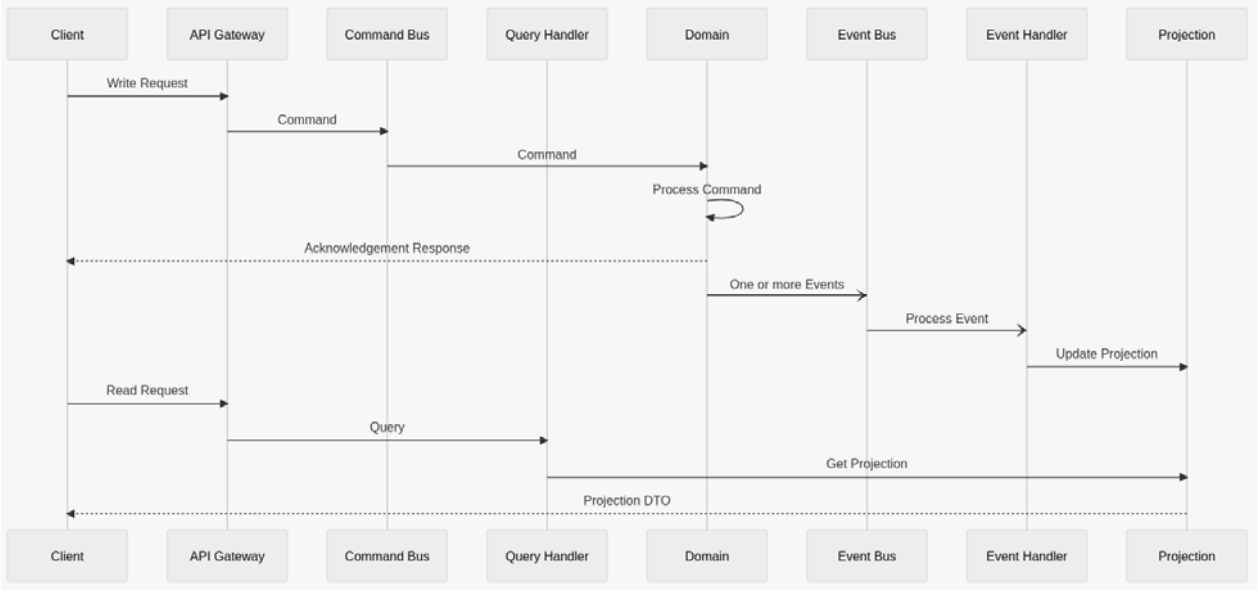

Figure 1 – The typical workflow of the command processing by the CQRS with ES system

DOI 10.15588/1607-3274-2024-3-11

After the request validation, the command enters a command handler, a certain component of the application business logic layer responsible for processing the request. The command handler uses repositories to retrieve an aggregate or some aggregates (domain layer entities) needed to perform the task. Then it calls a method or some methods of the aggregate [23]. In response, aggregate generates domain events which reflect the changes of the state of the aggregate.

Domain events are accepted and then put by the command handler to the Event Store unit responsible for saving and, publishing those events to Event Bus (the module responsible for delivery of the messages to subscribers, i.e. event handlers).

Event handlers subscribed to the different types of events receive and process published events. Some of the event handlers could be responsible for notification delivery, others for dynamic reports preparation (projections), etc.

Following [24] formally the system can be described as a tuple.

$$\left\langle H_c, H_e, \bigcup_{x \in H_e} S_x, P_y, \mathcal{N}, \mathcal{E}, E, T, \mathfrak{C}, O \right\rangle.$$

H_c is a set of command handlers responsible for processing incoming commands and publishing the notifications to event-delivery subsystem (EDS); H_e – a set of event handlers subscribed to notifications sent by the EDS; $\mathcal{E}$ – the set of all events occurred in domain; $\mathcal{N}$ – the set of all notifications – messages sent by all the publishers about the events; $E: \to \mathcal{E}$ – a unary function that maps a notification to the event the notification represents; $\mathfrak{C}$ – the set of event types related to a subset of events, and used to restrict the scope of variables, control the formation of expressions, and classify expressions by value [25]; $T: \mathcal{E} \to \mathfrak{C}$ – a function that maps an event to the event type, consiquently the notifications could also be mapped to event types using the composition of functions E and T, so we can say that $\mathfrak{C}$ represents the set of notification types as well; S_x – is a set of active subscriptions for event handler $x \in V$, where one subscription relates to a specific type of events; P_y – is a set of all notifications published by the command handler $y \in H_c$; O – a set of four basic interface operations [26] which can be defined as follows.

$pub_i^u(n_k)$ – notification $n_k \in \mathcal{N}$ related to event $e_k \in \mathcal{E}$ (which happened in domain) is published by the i-th command handler $w_i \in H_c$ to EDS at time u, that means that not all domain events obtained in result of command execution may be transformed to notifications and published by the command handler to EDS, but all the notifications are mapped to the events, $sub_j^t(A) \Leftrightarrow s_A \in S_{r_j}$ – event handler $r_j \in H_e$ can be subscribed to notifications about the events of a certain type $A \in \mathfrak{C}$ at time t. The result of the operation is the subscription added to a set of active subscriptions of the r_j, i.e. $s_A \in S_{r_j}$ Following [27] we can define the subscription $s_A \in S_{r_j}$ as a predicate: if the notification n_k matches the topic (or channel) of subscription (in our case the channel is related to notification type A), i.e. if $n_k : A$, then $s_A(n_k) \equiv \top$, and then the notification will be delivered to the event handler r_j at time $v > t$, denoted by $nfy_j^v(n_k)$, otherwise $s(n_k) \equiv \perp$, and the event will not be delivered to r_j.

$nfy_j^v(n_k)$ – is the operation of delivery of the notification. Thus, if j-th event handler is subscribed to notifications of A type, and the notification of that type is published by a publisher (we use '_' index to show the independency of command handler).

$$n_k : A \wedge pub_{\_}^u(n_k) \wedge s_A S_{r_j} \Leftrightarrow nfy_j^v(n_k), v > u. \qquad (1)$$

$usub_j^t(A) \Rightarrow s_A \notin S_{r_j}$ – event handler $r_j \in H_e$ can be unsubscribed from notifications about the events of a certain type A at time t, the result of the operation is the subscription excluded from the set of active subscriptions of the r_j.

This model does not reflect all the specific features of the system (e.g. events storing, replay mechanisms etc.) and is focused on the components connected with critical causal events issue.

It should be noted that inspire the difference of the meaning of the event and the notification terms we are intended to use only event term denoting the events passed by the command handler to EDS and then delivered to subscribers. Formally the problem can be described by the time diagram shown in Fig. 2.

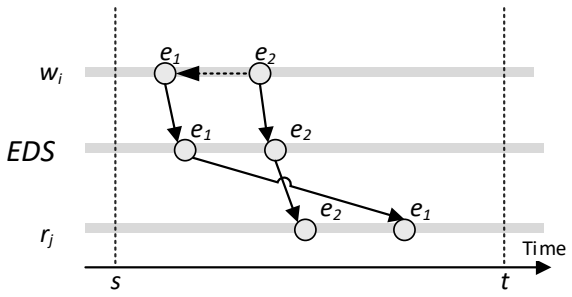

Figure 2 – A time diagram of a distributed computation

In the diagram command handler $w_i \in H_c$ in result of a command processing passes two events e_1, e_2 to the EDS, the dotted line shows that e_2 is an effect of e_1, i.e. e_1 is the cause of e_2. The projection of position of the events on the Time-axis shows their temporal relation as follows: $pub_i(e_1) \prec pub_i(e_2)$, i.e. publishing of e_1 by

DOI 10.15588/1607-3274-2024-3-11

w_i precedes publishing of e_2 by the same w_i command handler, "$\prec$" denotes strict partial order relation. Another way to denote the relation is $pub_i^u(e_1) \wedge pub_i^v(e_2)$, where $e_1, e_2 \in \varepsilon$, $s \le u \le v \le t$ and $s,u,v,t \in \mathcal{T}$, $\mathcal{T}$ is the set of the clock's ticks.

$e_1, e_2 \in P_i \bigcap I_j$, where P_i – a subset of the events published by w_i, I_j – a subset of incoming events which j-th event handler $r_j \in R$, is subscribed to, i.e. $P_i, I_j \subseteq \mathcal{N}$.

The interval of subscription $\tau_j(s_A, s, t) \in S_{r_j}$ of j-th event handler r_j to events of type A is an interval between subscription occurred at time s that can be denoted as $sub_j^s(A)$ and unsubscription occurred at time t denoted by $usub_j^t(A)$, s and t are timestamps, i.e. $s,t \in \mathcal{T}$ and $s < t$.

It means that if $e_k : A$ matches the subscription s_A in the defined interval τ_j, and the e_k is published by a command handler at time $s < u < t$, then the event will be necessarily delivered to the event handler r_j by the EDS and reversely, if the event is delivered then it was published by a command handler and matches the subscription in the defined interval. Formally this assertion can be described by the following formula.

$$e_k : A \wedge \tau_i \in S_{r_j} \wedge pub\_^u(e_k) \Leftrightarrow \Box\, nfy_j^{u'}(e_k), \qquad (2)$$

where $s \le u < u' \le t$ and $s,u,u',t \in \mathcal{T}$. We use the necessitation operator $\Box$ from modal logic in this formula to underline the restrictions of the model applied to the systems under consideration, because in different real systems the inviolability of the rule seems doubtful.

The formal description of the problem using modal logic [29] can be represented by following formula:

$$\tau_j^A, \tau_j^B \in S_{r_j} \wedge pub_i^u(e_1) \wedge pub_i^v(e_2) \Rightarrow \Box\left[\left(nfy_j^{u'}(e_1) \wedge nfy_j^{v'}(e_2)\right) \vee \left(nfy_j^{u''}(e_1) \wedge nfy_j^{v''}(e_2)\right)\right] \qquad (3)$$

where

$$\tau_j^A \equiv \tau_j(s_A, s, t), \tau_j^B \equiv \tau_j(s_B, s, t), e_1 : A, e_2 : B, \qquad (4)$$

$$s \le u < v < v' < u' \le t \text{ and } s \le u < v < u'' < v'' \le t$$
$$s,u,v,u',v',u'',v'',t \in \mathcal{T}.$$

This formula can be interpreted using Kripke semantics [30] as follows. For a model with worlds (states) w_1, w_2 accessible from actual world w, it is true that:

$$w_1 \models \tau_j^A, \tau_j^B \in S_{r_j} \wedge pub_i^u(e_1) \wedge pub_i^v(e_2) \Rightarrow \Box\left[nfy_j^{u'}(e_1) \wedge nfy_j^{v'}(e_2)\right] \qquad (5)$$

$$w_2 \models \tau_j^A, \tau_j^B \in S_{r_j} \wedge pub_i^u(e_1) \wedge pub_i^v(e_2) \Rightarrow \Box\left[nfy_j^{u''}(e_1) \wedge nfy_j^{v''}(e_2)\right] \qquad (6)$$

Which implies

$$w \models \tau_j^A, \tau_j^B \in S_{r_j} \wedge \left(pub_i^u(e_1) \wedge pub_i^v(e_2)\right) \Rightarrow \Box\left[\left(nfy_j^{u'}(e_1) \wedge nfy_j^{v'}(e_2)\right) \vee \left(nfy_j^{u''}(e_1) \wedge nfy_j^{v''}(e_2)\right)\right] \qquad (7)$$

$w, w_1, w_2 \in W$.

It means that for the system in w there could necessarily be one of the situations, i.e. accessible worlds w_1 and w_2, in which the events are delivered in the order they have been published (w_1 case) and – the reverse situation (w_2 case).

The consequence of this is:

$$w \models \tau_j^A, \tau_j^B \in S_{r_j} \wedge \left(pub_i^u(e_1) \wedge pub_i^v(e_2)\right) \Rightarrow \Diamond\left(nfy_j^{u'}(e_1) \wedge nfy_j^{v'}(e_2)\right) \qquad (8)$$

which means that the systems under consideration allow the situation when the causal events are not delivered in the order they have been published. Here the existential modality operator $\Diamond$ denotes the possibility of the situation.

It is worth to note that for some event handlers the order of handling causal events is not critical, and the interpretation of the identified problem mostly depends on the system configuration.

Let us see two examples of the projects where this issue is acutely expressed.

The first example is the clinic information system. The clinic specializes in surgery operations, including emergent surgery, but also provides consulting services. Each Hospitalization instance should refer to an instance of the Patient class, but in some cases (for example, emergent) the Hospitalization can be created as a result of the ResisterPatientWithHospitalization command execution. The appropriate command handler triggers the creation of the Patient aggregate and calling its AddHospitalization method. In result, command handler reads PatientCreated and HospitalizationCreated domain events from the Patient aggregate and passes them to EDS in <PatientCreated, HospitalizationCreated> order.

The application that uses the API reacts to the HospitalizationCreated event by checking the existence of the Patient instance and if it does not exist in a cache, it causes the error. But in the case of an EDS that does not guarantee the order, the events that were published in order <PatientCreated, HospitalizationCreated> can be delivered in order <HospitalizationCreated, PatientCreated> causing the error.

DOI 10.15588/1607-3274-2024-3-11

Another example is a financial system, where the broker is the owner of the group of users. Each broker must be connected with the group, but the group can be temporarily without an owner (this exceptional case can happen when the broker leaves the system for some reason). As a rule, the creation of the group is part of the broker creation process, but it could be the situation when the broker is assigned to the existing group which was owned by another broker. Thus, when a user adds the broker, the system can generate at least two variants of event sequences: <GroupCreated, BrokerCreated>, <BrokerCreated>. And, as in the previous example, it could cause an error in case of reverse order of delivered events.

Thus, this paper is devoted to resolving of these types of issues.

The solution of the problem of critical causal events synchronization depends on the solution of three main tasks which are as follows:

1) Providing a method of assessment of the likelihood of the issue which can be used to estimate the risks of critical causal events issue and to choose a proper strategy to address the issues discovered.

2) Providing a complex of methods and strategies to solve the issues considering the experience for handling the related problems in distributed information systems.

3) Providing a method of evaluation of effectiveness of the methods using the complexity and performance metrics.

2 REVIEW OF THE LITERATURE

The problem of synchronization of causal events was first addressed by L. Lamport [18]. In his paper he discussed the partial ordering defined by the "happened before" relation which is accepted by the researcher's community as the definition of "causality". In accordance with Lamport

Definition 1:

The "happened before" relation, in literature after Lamport denoted by →, is the smallest transitive relation that satisfies the following properties for any two events:

1) If a and b are events in the same process p_i, and a comes before b, then $a \rightarrow b$.

2) If a is the sending of a message (send event) by one process p_i and b is the receipt of the same message (receive event) by another process p_j, then $a \rightarrow b$.

3) If $a \rightarrow c \rightarrow \ldots \rightarrow b$ then $a \rightarrow b$.

Happened before is strict partial order relation ($\prec$) which is irreflexive, asymmetric and transitive [22]. Thus, $a \rightarrow b \Leftrightarrow a \prec b$.

Firstly, it is worth to note that Lamport wrote that this relation could be interpreted as that it is possible for event a to causally affect event b. Thus, the relation shows only causality potential, not true causality.

Secondly, the second property of the relation is very important for understanding the methods suggested to resolve problems of causality in distributed computing. According to [22] a send event reflects the fact that a message was sent; a receive event denotes the receipt of a message together with the local state change according to the contents of that message. A send event and a receive event are said to correspond if the same message that was sent in the send event is received in the receive event. It is also assumed that a send event and its corresponding receive event occur in different processes.

The presented semantics of event and a message term is slightly different from the semantics used in event-driven architecture, Publisher-Subscriber pattern [28], software systems, where the event is a signal emitted by a component upon reaching a given state, which carries an information about the fact of state changed and can be broadcasted to the processes subscribed to such type of events. The message contains a request or command, and it is point-to-point interaction oriented.

Lamport gave a distributed algorithm for extending it to a consistent total ordering of all the events which is based on logical clocks. Each process has its own clock function C_i that assigns a number (timestamp) $C_i(a_k)$ to event a_k in that process and $a \prec b \Rightarrow C(a) < C(b)$. Lamport's algorithm has well-known restriction $C(a) < C(b) \not\Rightarrow a \prec b$ to overcome which several approaches (e.g. vector clocks) were suggested [31, 32]. These methods relate to different restrictions and limitations. For example, in [22] assumed that each process is strictly sequential, and the events of the process are totally ordered by the sequence of their occurrence. According to [33] a set of vector timestamps, one per event, cannot fully characterize a distributed computation in the systems that allow message "overtaking". According to [34] these methods can only be used when the number of processes is known by every process, so each process can be assigned an integer number as an identifier.

Considering that causality is "cause-effect" connection of phenomenon through which one event (cause) under certain conditions gives rise to another event (effect) [22] and the true causality of events can only be denoted explicitly [35, 36].

There are two basic classes of methods connected with explicit definition of the events causality.

The first class of methods is based on using the causal history of events, which can be defined as follows [22]:

Definition 2:

Let $E = E_1 \cup \ldots \cup E_n$ denote the set of events of a distributed computation and let $a, b \in E$, $a \neq b$ denote events occurring in the course of that computation.

The causal history of b, denoted $H(b)$, is defined as $H(b) = \{a \mid a \in E, (a \xrightarrow{c} b)\} \cup \{e\}$. Where $\xrightarrow{c}$ denotes true causality relation. The projection of $H(b)$ on E_i, denoted $H(b)[i]$, is defined by $H(b)[i] = H(b) \cap E_i$.

DOI 10.15588/1607-3274-2024-3-11

It is worth to note that according to [22] E_i denotes the subset of events occurring at process p_i, while in the context of this work, E_i denotes a subset of events of a certain type, a subset defined by the characteristic predicate P_i, i.e. $E_i = \{a \mid P_i(a), a \in E\}$.

Definition 3:

Causality and causal history are related as follows:

$$1)\ a \xrightarrow{c} b \ \ iff\ a \in H(b). \tag{9}$$

$$2)\ a \| b \ \ iff\ a \notin H(b) \wedge b \notin H(a). \tag{10}$$

That means that causality relation is defined by the causality history.

The history of events can be represented differently depending on the specific of problems to be solved. For example, to solve the problem of causality tracking for distributed key-value stores in [36] proposed to describe the history by dotted version vectors etc. But the main idea is to attach causal history to the messages (in our case events) passing through the nodes-processes.

The simple way is to attach full causal history to each event, as it is proposed in [37]. Thus, the receiving process knows the order of events and waits for the completion of the causal event before reacting to the consequential one. The disadvantage of the method is the increasing of size of the additional information attached to the event. To reduce the size of payload a set of methods based on causal barrier has been proposed. These methods propose carrying information about the immediate predecessors of the event. This minimal information constitutes the so-called causal barrier of a message [34]. In [38][34] discussed a special type of causal broadcast, called Δ-causal broadcast, that can be used when the broadcast messages have a bounded lifetime Δ.

The second class of methods is based on an information protocol which allows to handle the causality of events specifying the information dependencies between the messages-communications that processes (agents [35]) may send. An agent may send a message-communication only if the state of the agent, which is based on the communication history, and the message-communication together satisfy the relevant information dependencies.

This approach is represented in works of Munindar P. Singh [35], who proposed a declarative, multi-agent approach based on true causality called information protocols. According to his works, the business process consists of a sequence of protocols, and each protocol reference must have at least one key parameter in common with the protocol in which its declaration occurs.

Another variant is to use communication protocols for point-to-point or multicast communications which enforce only a causal delivery order [39] that means that the delivery of messages has to be delayed according to causality constraints.

Among different types of distributed systems, the systems built using Publisher-Subscriber (Pub/Sub) pattern are the closest to the systems considered in this paper. Publishers and subscribers are interconnected by means of the so-called Notification Service, which plays a mediating role by storing the incoming subscriptions and routing incoming notifications towards the right destinations. For scalability reasons the notification service can be implemented in a distributed manner [40]. In [41] presented the basic operations classification, conditions and a Fault Model connected to Publisher-Subscriber systems which considers the problem of ordering.

In order to implement the causal order of published messages, in [28] apply causal barriers, but in comparison with the classical it does not enforce the causal order based on the identifiers of the nodes (per node vector) but by using direct message dependencies, which renders the algorithm more suitable for dealing with node dynamics. Thus, each *causal barrier[t]* keeps information on all messages that are predecessors of the next message that will be published by node *i* for topic *t*; the causal barrier consists thus of a set of message identifiers of format $\langle s, c \rangle$ (source and sequence counter).

In conclusion it should be highlighted that there are no well-known works connected with resolving the problem of critical causal events in the systems based on CQRS with ES architecture using Pub/Sub pattern to interact with the event handlers. The existing, described solutions depends on the specific of the problems they have been developed for, which makes it difficult to apply them directly to the problem under consideration.

The solution described in [35] has many similarities with the Saga pattern [42], suggested to address distributed transaction tasks and widely employed in information systems, for cases when modifying one aggregate leads to creating a command for modifying another aggregate. However, the considered issue arises when synchronizing causal events generated by changes to a single aggregate. While this approach could be applied in this situation, it would lead to suboptimal commands structure and performance degradation.

The solution provided in [37] is adapted to distributed virtual environment and cannot be directly applied to the systems under consideration. Additionally, it's not well understood how the authors resolve the problem with transitive events (i.e. $a \to c \to b$), and the case $a \to b \wedge a \to c \Rightarrow b \| b$ and $a \to c$ when the events *b* and *c* can be processed concurrently after the event *a* has been processed. Also, there is no information on how process sends the message to the process which is interested in only one event from the causal events package.

3 MATERIALS AND METHODS

The assessment of the likelihood of the causal events could help us not only to evaluate the risks connected to the issue, but also affects the choice of a method to manage the risks and to resolve the problem.

DOI 10.15588/1607-3274-2024-3-11

Based on research across several projects based on CQRS with ES architecture and Publisher-Subscriber pattern-based communication subsystem, conducted at DBB-Software company [43], where causal events issue was most acutely felt, the following dependencies has been identified.

It was noticed that the likelihood of the issue is contingent upon the interdependence of associated entities within the aggregate (Fig. 3) and the relation of the use cases connected with the entities. We preferably focused our attention on the aggregation relationships between the entities of 1–0..* and 1–1..* multiplicity as the most common sources of the causal events issue in the examined projects. It worth recalling that multiplicity in UML describes how many instances of one class can be connected to an instance of another class through a given association.

Let us consider the following example. In a clinical information system, a patient record can include information on several hospitalizations (at least one) and all the hospitalizations must be connected to a certain patient. In other words, the instance of patient aggregate must be associated with at least one instance of the hospitalization (association of 1–1..* multiplicity).

Thus, in result of CreatePatient command execution we may say that the patient creation event (PatientCreated) causes the hospitalization creation domain event (HospitaliztionCreated). On the other hand, a patient may not require surgery (i.e. the multiplicity of the "hospitalization-surgery" association is 1–0..*). In this case, the attached hospitalization information may not contain any information about surgery planning, etc. Thus, the likelihood of causal events in this case is rather lower than in the variant of 1–1..* association, but is not impossible.

To formalize the association, we use DLR description logic [44] which is the most suitable formalization mechanism (e.g. in comparison to [45]) for representing domain entities by means of concepts and relations. Description logics (DL) are regarded as late descendants of Minski's frames [29] with an explicit model-theoretic semantics. There are a number of automated reasoning systems, that have been successfully applied to various application domains.

According to [44] an aggregation A, saying that instances of the class $C1$ have components that are instances of the class $C2$, is formalized in DLR by means of a binary relation A together with the following assertion (Fig. 4):

$$A \sqsubseteq (1{:}C1) \sqcap (2{:}C2). \qquad (11)$$

Figure 3 – The typical example of relations between entities causing causal event issue

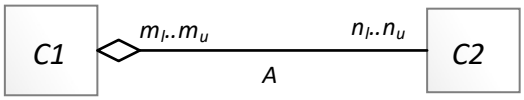

Figure 4 – Aggregation in UML [44]

The following convention is used: the first argument of the relation is the containing class ($C1$). The first component of the association is $C1$, the second is $C2$.

The multiplicity of an aggregation is expressed in DLR as follows.

$$C1 \sqsubseteq (\geq n_l[1]A) \sqcap (\leq n_u[1]A), \qquad (12)$$

$$C2 \sqsubseteq (\geq m_l[2]A) \sqcap (\leq m_u[2]A). \qquad (13)$$

If $n_\ell = 0$, i.e., the association is optional, the first conjunct could be omitted, and if $n_u = *$ (infinity) the second one is omitted.

Thus, using the example shown in Fig. 3. we can define following formulas.

$$Patient \sqsubseteq (\geq 1\ [1]hasHospitalization), \qquad (14)$$

$$Hospitalization \sqsubseteq (=1\ [2]hasHospitalization). \qquad (15)$$

where $=1\ [2]hasHospitalization$ is simplified variant of $\geq 1\ [2]hasHospitalization \sqcap \leq 1\ [2]hasHospitalization$

$$Hospitalization \sqsubseteq ([1]hasSurgery), \qquad (16)$$

$$Surgery \sqsubseteq (=1[2]hasSurgery). \qquad (17)$$

Before we start examining the use cases connected with the entities, we should formulate the restrictions and the specific of using terms and concepts.

Rigorously use case can be represented as a function which maps the request (in our case the request relates to command) into response and changes of the state of the system. Firstly, the function may map the request to different responses depending on the request content and the state of the system. Secondly, the function can be composed of other functions, considering different alternatives, i.e. $\lambda x.g(f(x)) : A \rightarrow C$ where $f : A \rightarrow B$ and $g : B \rightarrow C$. In accordance with Type theory [46] these functions can be described by the dependent functions (general productions) type denoted by $\Pi x{:}\ A.B(x)$, which means that if A is a type, we may have the family of types $B(x)$ where $x{:}\ A$.

Understanding the above specific, considering the works devoted to use cases formalization [47, 48], but guided by the purposes of the work trying to avoid unnecessary complexities connected to formulas representation, we should declare the following restrictions. Firstly, in this paper we are focusing only on the use cases related to the entities thus connected to three main operations (create, modify, remove). Secondly, we restrict the further using of the use case term to only the basic scenario omitting the exceptions, regarding only

DOI 10.15588/1607-3274-2024-3-11

conditional success-oriented alternatives (e.g. alternatives which may trigger the use case extension connected with the other entity).

Three basic use cases connected with the create operation for the entities shown in Fig. 3 are as follows: CreatePatient, CreateHospitalization, CreateSurgery (in real system the names of use cases can be different, considering the requirements dictated by the ubiquitous language used to design the software).

Table 1 – Relations between use cases

Relation	Meaning
$C(P) \xrightarrow{inc} C(H)$	$C(P)$ always invokes $C(H)$
$C(S) \xrightarrow{ext} C(H)$	$C(H)$ invokes C(S) only when a condition is met, not always
$C(P) \xrightarrow{pre} C(H)$	$C(H)$ cannot be invoked if $C(P)$ was not invoked before
$C(H) \xrightarrow{pre} C(S)$	$C(S)$ cannot be invoked if $C(H)$ was not invoked before

Let us denote CreatePatient use case as $C(P)$ (i.e. a successful path of the create patient scenario) CreateHospitalization as $C(H)$, CreateSurgery as $C(S)$.

The possible relations [49] between the use cases in accordance with the described model (Fig. 3) are presented in Table 1. To simplify the representation the relations are denoted as follows: the relation "$C(A)$ includes $C(B)$" as $C(A) \xrightarrow{inc} C(B)$, the relation "$C(B)$ extends $C(A)$" as $C(B) \xrightarrow{ext} C(A)$ the relation "$C(A)$ precedes $C(B)$" as $C(A) \xrightarrow{pre} C(B)$. It is worth to note that in according with terminology [50] in the relations "$C(P)$ includes/precedes $C(H)$" – $C(P)$ called the base use case, while in the relation "$C(H)$ extends $C(H)$" – the base use case is $C(H)$.

Let us denote hasHospitalization relation with the multiplicity in direction from *Patient* to *Hospitalization* as <*P, H*>+ (here "+" denotes 1..*) and the same association in opposite direction as <*H, P*>, analogically hasSurgery relation as <*H, S*>*, where "*" denotes 0..* multiplicity.

Let us denote PatientCreated event as e_p^c, HospitalizationCreated as e_h^c and SurgeryCreated as e_s^c. The relation $(e_p^c \xrightarrow{c} e_h^c)$ denotes that e_p^c is the cause of e_h^c.

The dependency of the likelihood of causal events on use cases is presented in Table 2. The likelihood is expressed using modal logic operators (□ – stands for necessary truth, ◊ – possibility, ¬◊ – imposibility of the situation).

Table 2 – Likelihood of causal events

Trigger	Use cases relation	Entities relation	Likelihood of causal events
$C(P)$	$C(P) \xrightarrow{inc} C(H)$	<*P,H*> +	$\Box (e_p^c \xrightarrow{c} e_h^c)$
$C(H)$	$C(S) \xrightarrow{ext} C(H)$	<*H,S*>*	$\Diamond (e_h^c \xrightarrow{c} e_s^c)$
$C(H)$	$C(P) \xrightarrow{pre} C(H)$	<*P,H*> +	$\neg\Diamond (e_h^c \xrightarrow{c} e_p^c)$
$C(S)$	$C(H) \xrightarrow{pre} C(S)$	<*H,S*>*	$\neg\Diamond (e_s^c \xrightarrow{c} e_h^c)$

Considering use cases connected with modification (denoted by *M* operation) and remove (denoted by *r* predicate) types of commands (for example, the activation of account belongs to the use case of *m*-type) the following types of relations among use cases should also be analysed:

1) Relations between creation use cases connected to one entity and modification use cases connected to other entities.

2) Relations between modification use cases connected to one entity and the creation use cases connected to other entities.

3) Relations between modification use cases connected to one entity and the modification use cases connected to other entities.

4) Relations between remove use cases connected to one entity and the modification use cases connected to other entities.

5) Relations between remove use cases connected to one entity and the remove use cases connected to other entities.

On the base of several projects analysis the following results were obtained. There are two generic necessary truth rules which can be represented as follows.

The first necessary truth rule considers creation of the instances of classes connected by $\geq 1[1]r$ type of relation.

Let us say *A* and *B* are the entities connected by *r*, and $C(A)$ and $C(B)$ are the use cases connected with the creation of their instances *a: A* and *b: B*, $C(A) \xrightarrow{inc} C(B)$ means that use case $C(A)$ includes use case $C(B)$, e_a^c, e_b^c are the events connected with the creation of the instances, $e_a^c \xrightarrow{c} e_b^c$ denotes that e_a^c is the cause of e_b^c, □ – stands for necessary truth. Then the first rule of the dependency between the relation of the entities, creation use cases and the corresponding causal events can be described by the logic formula as follows:

$$Iff\ A \sqsubseteq \geq 1\ [1]r,\ B \sqsubseteq = 1\ [2]r\ Then$$
$$C(A) \xrightarrow{inc} C(B) \Rightarrow \Box (e_a^c \xrightarrow{c} e_b^c) \quad (18)$$

Consequently

$$Iff\ A \sqsubseteq \geq 1\ [1]r,\ B \sqsubseteq = 1\ [2]r\ \ Then\ \ \Box (e_a^c \xrightarrow{c} e_b^c) \quad (19)$$

DOI 10.15588/1607-3274-2024-3-11

The second necessary truth rule considers removal of the instances of classes. Independently of the first component of the relationship, if the multiplicity of the second is =1, then the remove use cases will be connected with includes relationship and cause the events causality, which can be described by the logic formula as follows
Iff $B \sqsubseteq = 1$ [2]*r Then*

$$R(A) \xrightarrow{inc} R(B) \Rightarrow \Box (e_a^r \xrightarrow{c} e_b^r) \qquad (20)$$

$R(A) \xrightarrow{inc} R(B)$ means that use case $R(A)$ includes use case $R(B)$, e_a^r, e_b^r are the events connected with the removing of the instances.

The other relations between the use cases depend on the functional requirements of the system and cannot be generalized, but they could significantly affect the risk of the issue, therefore we tried to organize the information which was obtained from the realized projects using relationship matrices shown in Fig. 5 and Fig. 6.

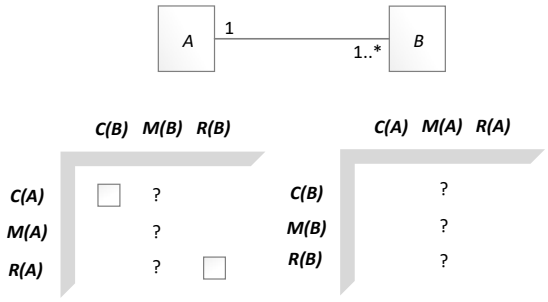

Figure 5 – Dependency of causal events on the relations between the use cases for 1–1..* relation

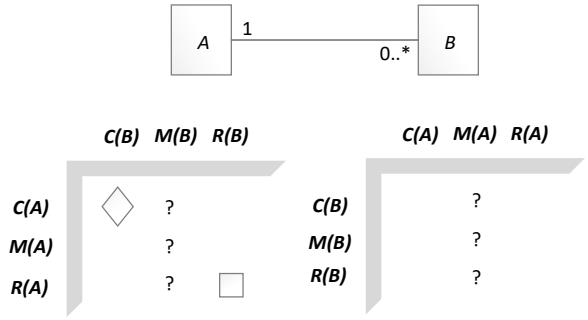

Figure 6 – Dependency of causal events between the use cases for 1–0..* relation

The left matrix shows the relations in direction from the use cases engaging A entity, to the use cases engaging B one, i.e. $<U_i(A), U_j(B)>$ pairs, where B is the aggregated entity, the right matrix shows the relationships of the use cases in opposite direction. $U_i(X)$ and $U_j(X)$ stand for one of $\mathfrak{A}$ use case types (the basic types are C, M, R), engaging X entity. $\Box$ – stands for necessary truth, $\neg\Diamond$ – possibility, empty space means that the relation is impossible, i.e. $\neg\Diamond$ necessary false. "?" – means that the likelihood is undefined because it depends on the specific of the system requirements and whilst for one subset of the existing systems under consideration it may be necessary truth, for second it may be only possible and for the third it is absolutely impossible. For example, if, in accordance with the requirements, the relation between $M(A)$ and $M(B)$ is "$M(A)$ includes $M(B)$" then the likelihood of the causal events is necessary truth, i.e. $\Box (e_a^m \xrightarrow{c} e_b^m)$ etc.

Some of the presented dependencies are not trivial and connected to the certain type of the projects. For example, the relation $<C(A), C(B)>$ (Fig. 6) is rear: it appeared in the financial system where $C(A)$ caused the modification of several existing instances of B in accordance with the settings rule, which can then be aggregated by the object.

The generic use cases diagrams for two basic variants of the entities relations mentioned above are shown in Fig. 7. The dotted edges without labels denote undefined relations.

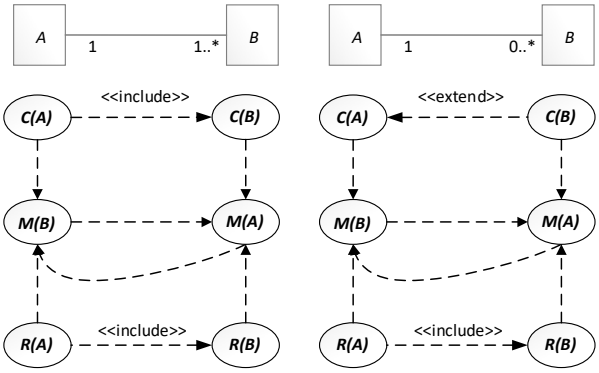

Figure 7 – Relations between basic use cases for 1–0..* and 1–1..* associations

Undefined dependencies for the certain project can be resolved using the information on the projects (ideally of the same type) which has been previously realized.

The probability of the include or extend type relation occurrence can be evaluated by the ratio of frequency of occurrence of such relation to the number of aggregation relationships within the project.

In result, we can evaluate the number of potential cases where the issue could appear, and, consequently, to choose the method of its resolution.

For example, if we have the probabilities of the include-type relation between the use cases connected with A and B entities linked by the association of 1–1..* multiplicity for a project of a certain type as it is shown in Fig. 8. Then we can guess that the number of the potential causal events for each aggregation relation with the 1–1..* for a new project of the same type will be approximately 2.9 (i.e. for 10 cases it will be 29 etc.). The same way we can evaluate the number of cases for the relation of the extend-type.

To increase the accuracy of the estimation not only projects should be classified, but also the entities and their relations should be also considered.

Of course, the evaluation cannot identify the number of critical causal events. To evaluate the potential number of critical causal events we may monitor the ratio of their

DOI 10.15588/1607-3274-2024-3-11

number to the number of the inclusion and extension probabilities.

Thus, the result may be as it is shown in Fig. 9.

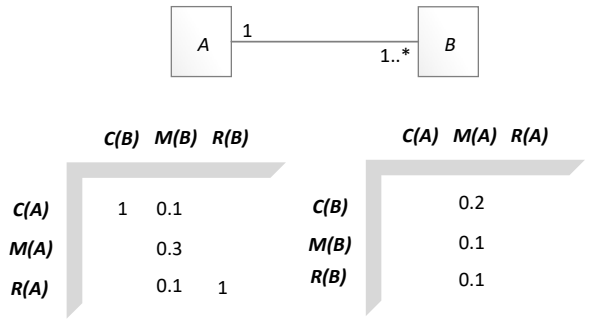

Figure 8 – The probabilities of the include-type relation between the use cases for a project

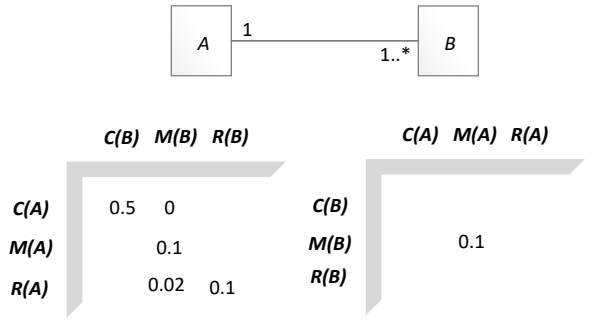

Figure 9 – The probabilities of the potential number of critical causal events for include-type relation between the use cases for a certain type of entities

This approach can give a more or less accurate assessment of the probability of potential critical causal events for a project if the company has already implemented similar projects using the same architecture. It should be noted that the classification of projects, entities, and maintaining statistics on projects is a labour-intensive task and cannot be implemented without automation.

In conclusion, to summarize the presented information we can define as follows. The best way to estimate the risk of critical causal events occurrence is to use the history related to the same type of projects, considering relations of 1–1..* and 0–1..* types between the entities and the derived relations among use cases. Whether the absence of the history it is very important to take into consideration the dependencies of causal events on the relations between the use cases shown in Fig. 5 and Fig. 6 and thoroughly analyse the project trying, firstly, to choose the entities linked by the relations of such type, secondly, to analyse the use cases identifying the causality of the events, and, thirdly, choosing critical causal events of them.

Depends on the likelihood of critical causal events, different solutions are favourable. Let's consider four solutions to the issue of synchronizing critical causal events in systems based on CQRS with ES architecture, along with their advantages and disadvantages.

The first variant is "New event introduction". The main idea is to create a new event that represents the composition of critical causal events.

Let us see the modifications of the system in case when the use cases connected by the "includes" association.

$$U_i(A) \xrightarrow{inc} U_j(B) \wedge U_i(A) \Rightarrow \Box(E_{a+b}^{U_i}) \qquad (21)$$

In this case $U_i(A)$ triggers execution of $U_j(B)$ use case which results in generating one composed event $\Box(E_{a+b}^{U_i})$ which can be linked to U_i use case (e.g. PatientWithHospitalizationCreated which contains included HospitalizationCreated event). At first glance the number of events remains the same (just PatientCreated event is substituted by PatientWithHospitalizationCreated), but the handler subscribed to $E_b^{U_j}$ type of events should be also subscribed to the events of $E_{a+b}^{U_i}$, because in case when $U_j(B)$ occurs independently of $U_i(A)$, which can happen in case of 1–1..* relation, an event of $E_b^{U_j}$ will be generated, i.e.

$$U_i(A) \xrightarrow{inc} U_j(B) \wedge U_j(B) \Rightarrow \Box(E_b^{U_j}) \Rightarrow \neg\Diamond\neg E_{a+b}^{U_i}. \qquad (22)$$

In case when the use cases connected by the "extends" association the situation is as follows.

$$U_j(B) \xrightarrow{ext} U_i(A) \wedge U_i(A) \Rightarrow \Box(E_{a+b}^{U_i} \vee E_a^{U_i}) \qquad (23)$$

and

$$U_j(B) \xrightarrow{ext} U_i(A) \wedge U_j(B) \Rightarrow \Box(E_b^{U_j}). \qquad (24)$$

Thus, three events should be introduced and all $E_b^{U_j}$ and $E_a^{U_i}$ handlers should be also subscribed to $E_{a+b}^{U_i}$ event.

The situation becomes worse when we have the following relation of the use cases (Fig. 10).

In this case $U_i(A)$ can result four different types of events.

$$\left[U_j(B) \xrightarrow{ext} U_i(A) \wedge U_k(C) \xrightarrow{ext} U_i(A)\right] \wedge U_i(A) \Rightarrow \Box(E_{a+b+c}^{U_i} \vee E_{a+b}^{U_i} \vee E_{a+c}^{U_i} \vee E_a^{U_i}). \qquad (25)$$

DOI 10.15588/1607-3274-2024-3-11

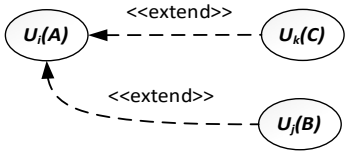

Figure 10 – Multiple extend relation example

$E_a^{U_i}$ handler should be subscribed to and consider handling of $E_{a+b+c}^{U_i}, E_{a+b}^{U_i}, E_{a+c}^{U_i}$ types of events.

$E_{a+b}^{U_i}, E_{a+c}^{U_i}$ handlers should consider $E_{a+b+c}^{U_i}$.

$E_b^{U_j}$ handler to $E_{a+b+c}^{U_i}, E_{a+b}^{U_i}$.

$E_c^{U_k}$ handler to $E_{a+b+c}^{U_i}, E_{a+c}^{U_i}$.

For example, there is HospitalizationCreated event created as a result of processing the command. It can optionally cause the creation of surgery and diagnostic procedures. Such case can lead to creation of three composite events:

– HospitalizationWithSurgeryAndDiagnosticCreated;
– HospitalizationWithSurgeryCreated;
– HospitalizationWithDiagnosticCreated.

HospitalizationCreatedHandler should probably need the subscription to all three of them, HospitalizationWithSurgeryCreatedHandler and HospitalizationWithDiagnosticCreatedHandler potentially need the subscription to HospitalizationWithSurgeryAndDiagnosticCreated. SurgeryCreatedHandler and DiagnosticCreatedHandler both need subscription to HospitalizationWithSurgeryAndDiagnosticCreated as well as HospitalizationWithSurgeryCreated and HospitalizationWithDiagnosticCreated respectively. So, using this synchronization method with such a relationship can generate 8 additional subscriptions for existing handlers to new events, increasing code complexity and deteriorate code readability.

Now, let's imagine that there are 10 occurrences of such three use cases relations. It results in introduction 30 extra events, and 80 new subscriptions, as well as extra work related to updating the handlers. Thus, the provided solution can lead to an increase in code complexity and often results in code duplication [51].

Of course, here is presented the worst-case scenario. If the $a + b$ and $a + c$ combinations of events are not critical, these types of events can be ignored and only one $a + b + c$ extra type of events must be added.

The second variant of the problem's solution is based on using synchronous event queues instead of a classical event bus variant [52, 53]. It solves the issue by ordering the handling of events, but it may cause performance issues.

The third variant is the variation of Causal Barrier method [28] [38], the main idea of which is to provide partial history of causality considering the bounded lifetime Δ for the events-messages [34]. This method is effective when the number of causal events in the history can be high, which can negatively affect the performance of the event-driven system. The other assumption connected to that method is that handling of the events may not require full history of causality. But for the systems under consideration the history of causal events does not exceed 3–4 events and dividing the history into chunks results in handlers' complication and decreasing the usability of API (when the handlers are the third-party services). The cases of handling events by the subscribers using only partial history are the exception rather than the rule. As it is mentioned in [19], to maintain the causal order of events, a site must verify that all events within the causal history of the received event have been handled before processing it.

The complexity of the modification can be expressed in terms of the number of scenarios that the client needs to handle depending on the number of causal events within the *Ej* group.

The modifications that should be applied to each handler for two causal events are shown in Fig. 11.

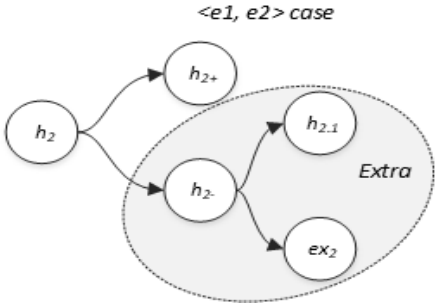

Figure 11 – The modifications must be applied according to the Causal Barrier approach to process two causal events

For the case of two causal events the following notation is used:

$< e_1, e_2 >$ – an ordered set of causal events, where $e_1 < e_2$ ("<" means the strict partial order relation).

h_{2+} – basic-positive part of the handler responsible for processing E_z according to the scenario when the events come in proper order i.e. $< e_1, e_2 >, e_1 : E_1, e_2 : E_2$. This part cannot be omitted and can be regarded as a minimal part of the handler needed to process a e_2 event.

h_{2-} – alternative-negative part of the handler responsible for processing the following situations:

$h_{2.1}$ – when the order of events $< e_2, e_1 >$ instead of $< e_1, e_2 >$;

ex_2 – an exceptional situation when e_2 is lost or can be considered as lost after a defined period of time, i.e. bounded lifetime Δ has expired.

DOI 10.15588/1607-3274-2024-3-11

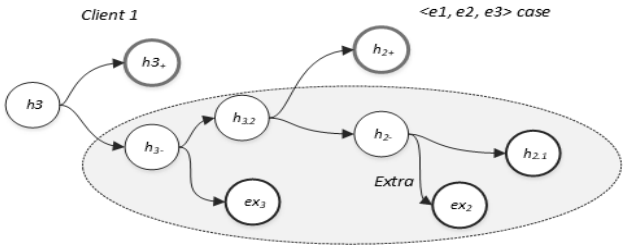

Figure 12 – The modifications must be applied according to the Causal Barrier approach to process three causal events

The three causal events case (Fig. 12) looks much more complicated. The case h_{2-} should be considered, when e_2 has come but e_1 has not come yet, and e_2 is in the state of waiting for confirmation. It is notable that the current description doesn't touch the part responsible for exception handling which can be realized in different ways.

Thus, the number of scenarios that the client needs to handle (the complexity of the modification) depending on the number of connected events within the *Ej* group can be evaluated using the formula (26).

$$\left|C(Ej)\right| = 2*k-1. \quad (26)$$

Let us see the variant of client modification in the example for the sequence of events <PatientCreated, HospitalizationCreated>.

The first scenario is when the sequence of events is received by the client application in the order as it was sent, e.g. in the proper order. In this case, the events are processed sequentially (see Fig. 13).

The second scenario involves a situation where the HospitalizationCreated notification arrives first. In this case, after receiving the HospitalizationCreated notification and determining that the patient to which the hospitalization belongs does not exist, the client waits for a PatientCreated event for a specified period. Upon receiving the PatientCreated notification, both notifications are processed together (see Fig. 14).

In the last scenario, if the PatientCreated notification does not arrive within the specified waiting time, the Client logs an error and/or requests a full initial context from the server (see Fig. 15).

This solution solves the problem, but it significantly complicates the construction of the client, negatively affecting the usability of the system's API, i.e. each client such as different mobile and desktop applications, including other services should be prepared to handle these cases.

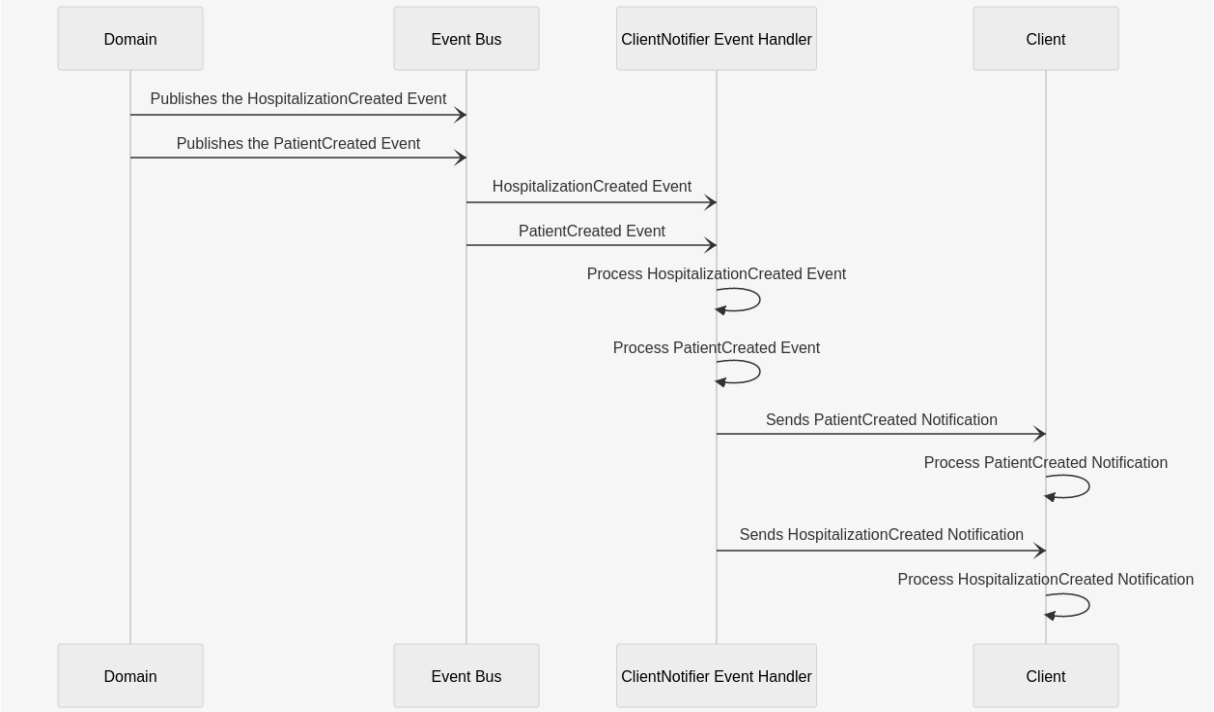

Figure 13 – The basic scenario of events processing using the Causal Barrier approach. The case when the PatientCreated event is received before HospitalizationCreated

DOI 10.15588/1607-3274-2024-3-11

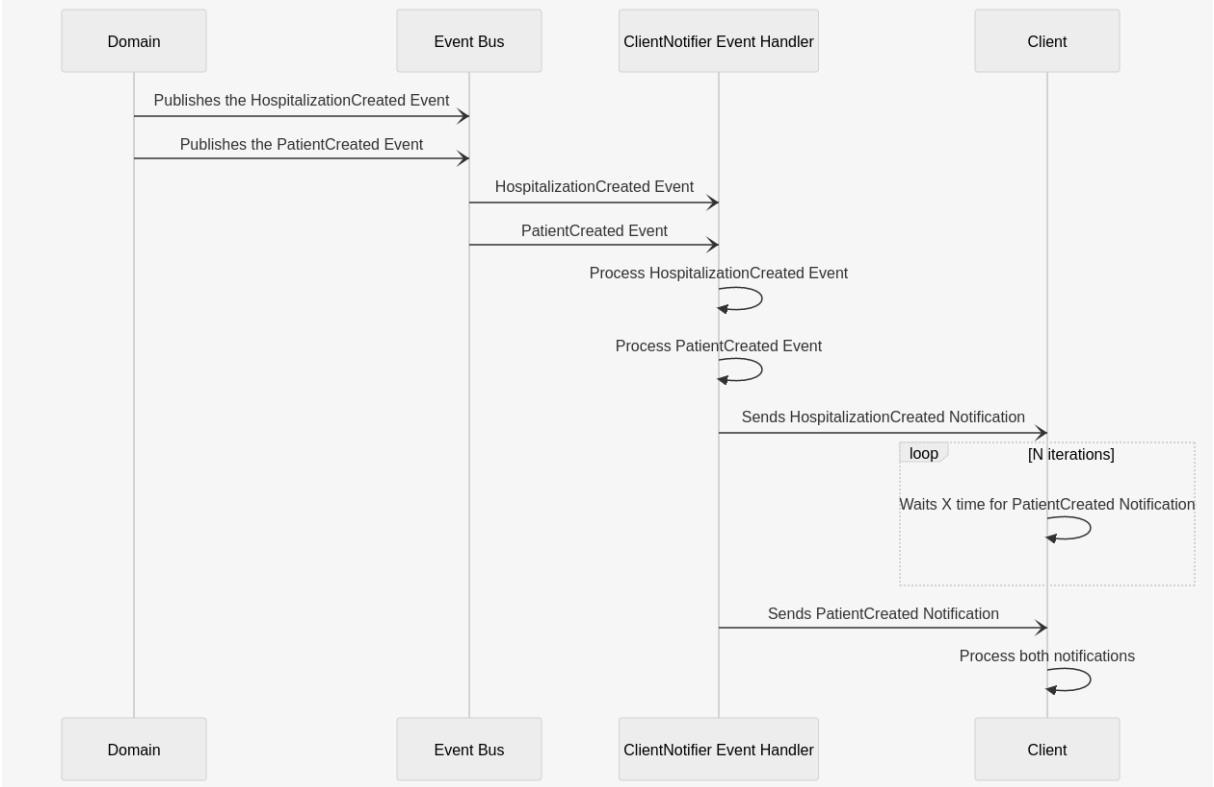

Figure 14 – The basic scenario of events processing using the Causal Barrier approach. The case when the PatientCreated event is received after HospitalizationCreated

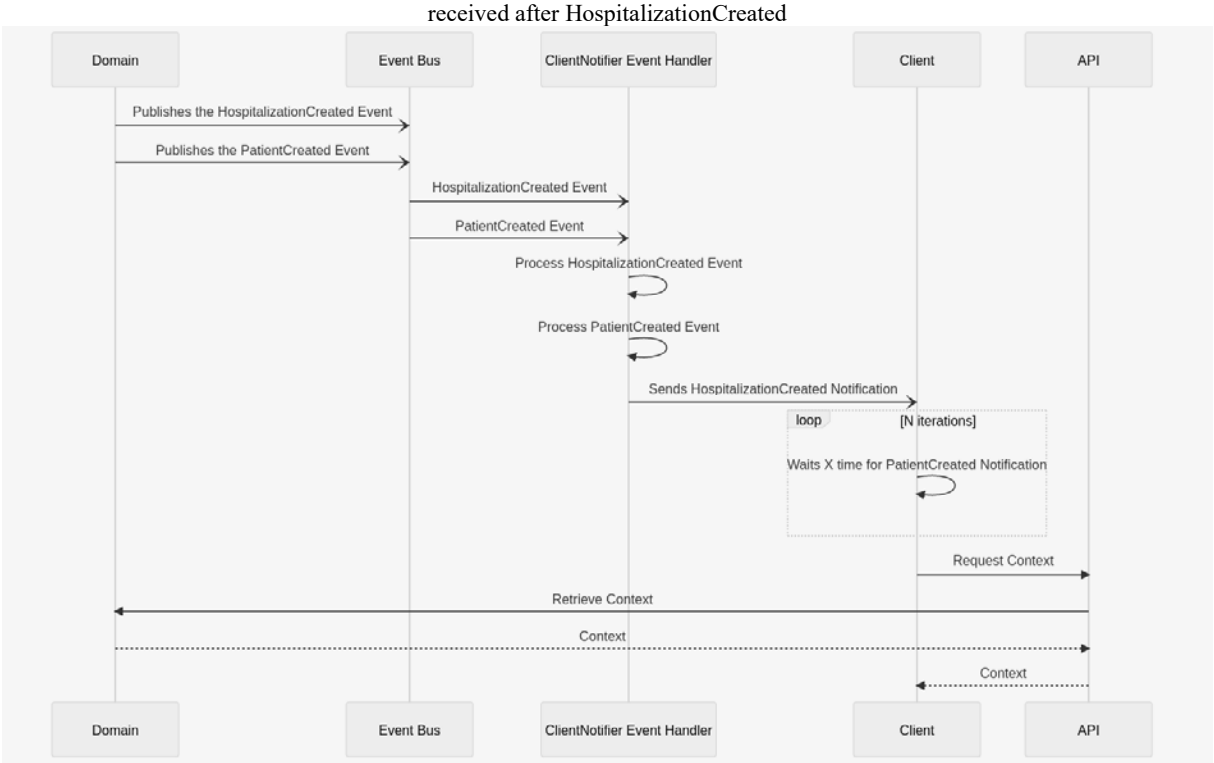

Figure 15 – The basic scenario of events processing using the Causal Barrier approach. The case when the PatientCreated event is not received

The last variant of the solution is based on presenting of full history of causality. The variation proposed in this paper differs from the classical solutions (e.g. [37]) in that it is more flexible and more effective for CQRS with ES architecture.

The solution is based on introduction of an abstract container of events that can be used for delivering the

DOI 10.15588/1607-3274-2024-3-11

causal events. It can be called a complex event or a Container of Events (Code snippet 1).

There are three basic ways of event publishing unit modifications in order to deliver event containers. The first one which is based on Event Bus modification is as follows.

For each container, Event Bus (i.e. event publishing unit) analyses its content and if some handler is subscribed to one or more events from the group, it delivers these events in proper order to that handler. This variant requires spending more effort to modify the event publishing unit, but it does not touch the handlers.

The second variant, which is the opposite variant to the first one, is to deliver all event containers to all the event handlers (a variant of broadcast notification) making the handlers responsible for analysing the content of containers and choosing the right method to process the events. This variant requires the introduction of an abstract handler able to get the events from the container and choose the proper method/methods of their processing. The advantage of the solution is avoiding Event Bus modification.

The third variant is the composition of the first and the second. Instead of doing container of events broadcasting, Event Bus sends the event containers only to the event handlers subscribed to one or more events from the group, using a generic method. In this case, the event handler is responsible for analysing the payload, defining the order of events publishing, and choosing the proper method/methods to process the events. This variant seems to be the most effective solution, because of reducing the number of handlers to notify, but it requires a slight modification of the Event Bus.

At the implementation level, for the third variant, two following approaches should be considered. The first one involves creating a container event handler within the base class, which unpacks the container of events and then invokes the appropriate methods of the derived classes, passing subgroups of events from the container. This approach is based on defining method signatures, using reflection mechanisms. It does not require any modifications to the handle methods of the derived classes and proves effective when integrating Container of Events solutions into a system that already has a large number of handlers. The second approach entails defining in the base class only the function for unpacking the container of events, which is then utilized in the handle methods of the subclasses. After unpacking, these methods process events in the defined order. This solution is more flexible, as it allows adding additional logic for handling events section of handle methods of each handler.

The complexity of this solution is independent of the number of causal events within the *Ej* group. All the necessary changes are made to the system's infrastructure, and it plays a crucial role in assessing the ease of implementing future changes in the system across observable solutions.

To reduce resource costs when implementing new handlers, a base handler with an implementation of the UnpackContainer function is created (Code snippet 2) All handlers processing causal events inherit from the BaseHandler (Fig. 16). Upon receiving the event container, the handler unpacks it and processes the events synchronously (Code snippet 3).

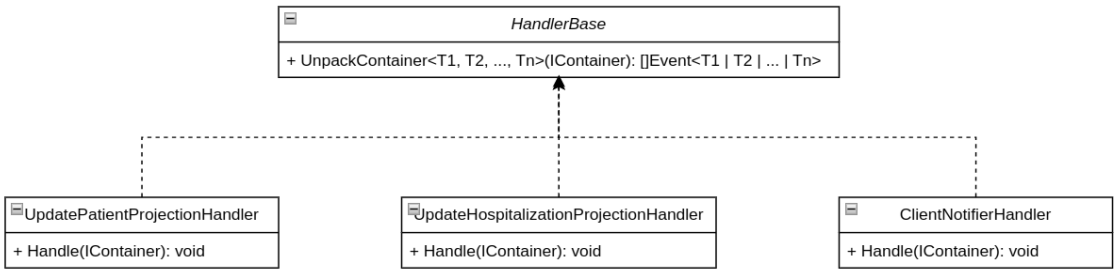

Figure 16 – The variant of solution based on the introduction of the BaseHandler.

Code snippet 1

```
{
    Title: string,
    Events:  [
            {
                header:  string,
                body:            JSON string
            }
    ]
}
```


DOI 10.15588/1607-3274-2024-3-11

Code snippet 2

```
class HandlerBase {
  UnpackContainer<T1, T2, ..., Tn>(container: IContainer): []Event<T1 | T2 | ... | Tn> {
                events: []Event<T1 | T2 | ... | Tn> = []
                For container.Events (event: Event) {
                        if ([T1, T2, ..., Tn].includes.(event.type) {
                                events.Add(events)
                        }
                }
                return events
        }
}
```

Code snippet 3

```
class Handler inherits HandlerBase {
        Handle(container: IContainer): void {
                events = self.UnpackContainer<_handledTypes>(container)
                For events (event: Event<_handledTypes>) {
                        Process event synchronously
                }
        }
}
```

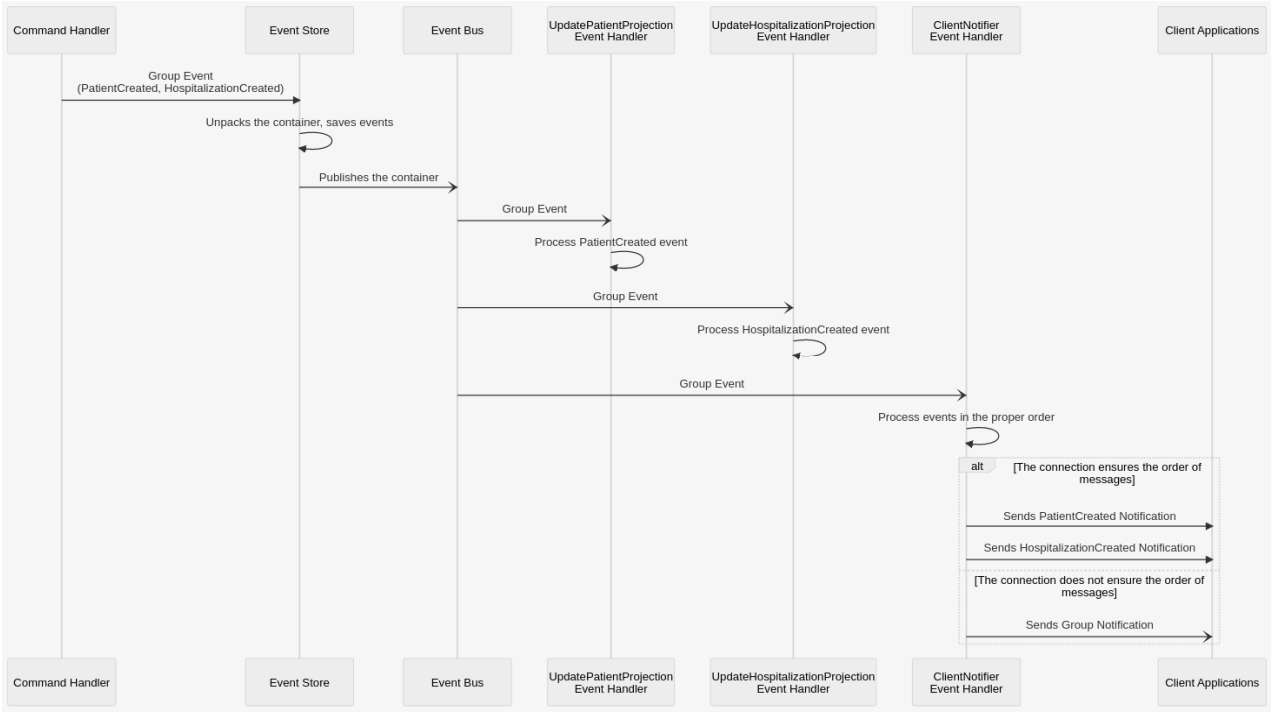

Figure 17 – The basic scenario of events processing according to the Container of Events approach

For the above example, it works as follows. For the hospitalization creation task, the command handler puts the sequence of generated events into a container with the title "HospitalizationCreation" (<PatientCreated, HospitalizationCreated>). Then it passes the container to the Event Store which unpacks the container, saves events, and publishes the container. Then the event handlers subscribed to the PatientCreated or HospitalizationCreated events receive the container. In our case UpdatePatientProjection handler is subscribed to PatientCreated event, UpdateHospitalizationProjection – to HospitalizationCreated event and ClientNotifier handler – to both of them. So all these handlers will receive the container, unpack that container and process the events in proper order which is <PatientCreated, HospitalizationCreated> (see Fig. 17). After processing

DOI 10.15588/1607-3274-2024-3-11

the Container of Events by Notification handler, the Notifications are sent to the Client Application. If the connection between the Notification Service and the Client Application guarantees that the order of delivery is preserved, the service sends notifications one by one in the correct order (e.g. WebSockets guarantees the preservation of the delivery order [54]). Otherwise, the Notification Service builds a notification, that includes information from container of events, and sends it to the client application as a single message. In that case, the client application should be adapted for notifications handling making the client application able to transform the Container of Events into a sequence of events calling them one by one in the correct order.

4 EXPERIMENTS

The strategy chosen to conduct the experiment is as follows.

The typical test application to realize the tasks described in part 3 was created and published to GitHub [55]. Initially, the system domain contained only two causal events that were processed asynchronously. The four solutions proposed above were realized concurrently and published to different Git branches.

The first phase of the experiment involves evaluating the complexity of the code required for integrating each of the four modifications.

In the second phase of the experiment two additional causal events were added to the domain of the system, and these changes were merged into each version with the integrated methods for solving the causal events synchronization issue. Then, after merging, the versions were updated to be operational and the complexity of code changes for the update was evaluated for each method.

The third phase of the experiment involves evaluating the performance of each of the integrated methods.

The fourth phase can be considered as calculating the complexity-performance comparative assessment of the considered methods.

Several methods for assessing the complexity of task implementation were considered [56][57]:

Lines of Code [57]. The approach suggests that the complexity of a software product is directly dependent on the number of lines of code in the product. It's a simple but not very accurate and relevant estimation.

The Number of Statements Metrics [56] serves as an indicator of the quantity of statements within a method. On the positive side, the Number of Statements Metrics offers a nuanced measure of method complexity, providing a more stable evaluation compared to Lines of Code. It encourages the identification of logical groupings within a method, fostering improved code organization. However, a potential drawback lies in its reliance on the subjective process of method extraction, which could introduce variability in interpretation.

Object Points [57] is a metric method that assigns weights to software modules. While it can be described, the process of assigning weights may lack clarity, and it does not inherently consider the uniqueness of the code. This metric primarily serves estimation purposes rather than evaluating the finalized code.

Information flow complexity [58]. The method entails evaluating information flow complexity in a software system through the analysis of function call quantity, frequency of invocation, and the number of functions each function calls. Its robustness lies in offering a holistic perspective on the data and control flow within the system, facilitating the identification of dependencies and potential bottlenecks. This approach is especially valuable for assessing and managing the complexity of software codebases characterized by considerable function nesting.

Cognitive functional size [59]. The Cognitive Functional Size (CFS) approach involves quantifying the functional size of software based on the cognitive load required for developers to comprehend and interact with the system. Notably, it excels in providing a user-centric measurement, capturing the complexity from the perspective of understanding and processing functionality. This method is particularly valuable for comparing the complexity of entities, such as classes, offering a more insightful evaluation that aligns with the cognitive demands placed on developers interacting with those entities.

Dep-degree metrics [60]. The approach operates on the principle that a program becomes more challenging to understand as the programmer's short-term memory is burdened with more chunks to remember. The DepDegree is a method of the cumulative count of dependencies for its statements, aligning with the psychological understanding that immediate memory has a limited capacity.

McCabe's cyclomatic complexity [61]. The complexity measurement is based on the amount and level of functions, methods, and procedures (e.g. loops and conditions). The higher this amount, the more difficult it will be for the developer to build, understand, and modify the code. This method excels most when evaluating complex algorithms. For simple operations, it may not provide high accuracy and may not reveal the true variability in the complexity of implementation.

Halstead Software Science Metrics (HSSM) [62]. These metrics are used to quantify the complexity of software by analysing the composition of code within program modules. The approach calculates three primary complexity metrics of a program: volume (V), difficulty (D), and an effort (E). The formula for calculating the effort in Halstead Software Science Metrics is as follows:

$$E = D * V. \tag{27}$$

V represents the program's volume, which is calculated using formula 28. D represents the program's difficulty, which is computed using formula 29.

$$V = (N_1 + N_2) * \log_2(n_1 + n_2). \tag{28}$$

$$D = (n_1 / 2) * (N_2 / n_2). \tag{29}$$

DOI 10.15588/1607-3274-2024-3-11

The Halstead Software Science Metrics is a straightforward method for measuring code complexity that performs well in assessing the intricacy of simple code. It takes into consideration both the volume of code and its uniqueness. The main cons of the method are ignoring the higher-level software design and architectural considerations and limited scope. Halstead Metrics primarily focuses on the code itself and may not provide a comprehensive view of software code quality, performance, or other important factors.

Another critical metric in such systems, besides complexity and the effort required to implement a solution, is performance. The performance metric was determined by measuring the time taken, experimentally, from the submission of a data change request to the data update on the Client side.

To provide an overview and highlight the pros and cons of each approach, three different measurements were conducted:

– Average update time when sending 100 data update requests with a 10-millisecond interval.

– Average update time when sending 1000 data update requests with a 10-millisecond interval.

– Average update time when sending 100 parallel data update requests, repeated 100 times with a 200-milliseconds interval.

For a more accurate assessment, the experiment was repeated three times using machines with different technical specifications. The final evaluation is the arithmetic mean of the three obtained measurements. It is also worth noting that a relatively simple implementation of the considered solution methods was provided for the experiment. For example, for the variant with a queue, the in-memory queue "Sync-Queue" [63] was used; using other tools such as AWS SQS [64] or Apache Kafka [65] would yield different assessment results.

To compare this metric across multiple solutions, a specific system runtime metric was measured for each solution multiple times and the average value was calculated. Thus, this average value is a percentage relative to the maximum performance solution variant (formula 30).

$$P_k = \frac{T_{\min}}{T_k}. \tag{30}$$

To make the complexity-performance comparative assessment of the considered methods more descriptive, formula 31 is derived. When using this formula, the effort expended on integrating the solution (C_i) is considered equally important to the effort expended on the system's maintenance (C_m) with the integrated solution. Therefore, the coefficients α and β are set to 0.5. In other situations, the coefficient selection may involve using the Rank Correlation method [66]. Since the comparison is conducted within the scope of a single system, the significance of performance relative to implementation complexity can be disregarded, and the coefficient ρ = 1.

$$E_{\text{int}} = \rho * \frac{P_{avg}}{\alpha * C_i + \beta * C_m} * 10^l. \tag{31}$$

The effort for integration (C_i) is calculated using the Halstead Software Science Metrics method in Phase 1 of the experiment. The effort required for system's maintenance (C_m) calculated in Phase 2 of the experiment. For the average relative performance (P_{avg}) the arithmetic mean is taken among the three metrics obtained during the third phase of the experiment (formula 32).

$$P_{avg} = \frac{P_{ll} + P_{hl} + P_{hpl}}{3}. \tag{32}$$

Given that the relative performance is a percentage metric, and the effort calculated using the Halstead Software Science Metrics method has the order of four, let us assume the order of magnitude coefficient (l) to be 3.

5 RESULTS

Phase 1. The Halstead Software Science Metric provides a reasonably accurate reflection of the time spent on implementing each solution variant. The code of each solution was analysed and the number of distinct operators and operands (Distinct operators, Distinct operands, Occurrences of operators, and Occurrences of operands) were obtained. The metrics required for each variant implementation (Volume, Difficulty, and Effort) were calculated by formulas 27–29. The results of the calculation are represented in Table 3 and visualized in charts (Figs. 18–20).

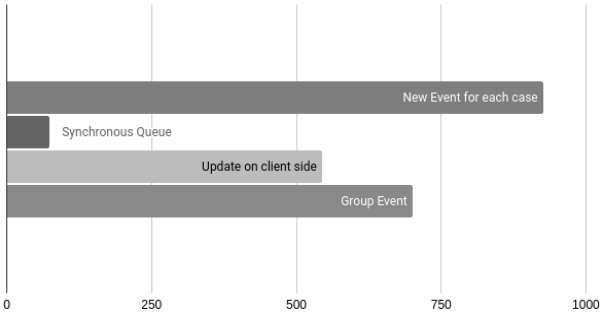

Figure 18 – Program Volume. Phase 1

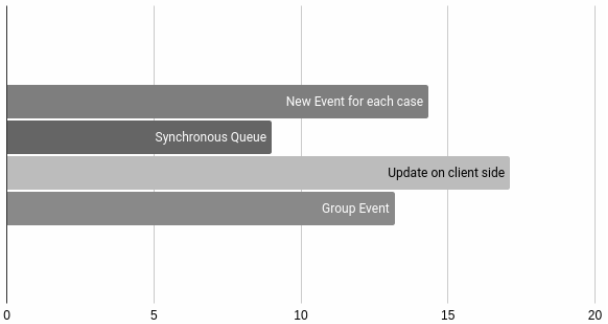

Figure 19 – Program Difficulty. Phase 1

DOI 10.15588/1607-3274-2024-3-11

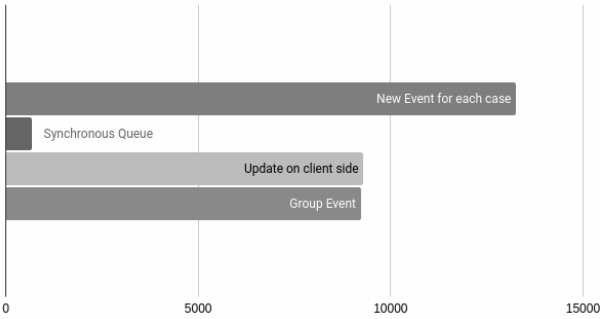

Figure 20 – Programming Effort. Phase 1

Phase 2. The second phase of the experiment was conducted to assess the additional efforts required for adding new pair of critical causal events to the system. I.e. the provided solution is already implemented, and new causal events are introduced. Similar to the previous phase of the experiment metrics are calculated and represented in Table 4 and Fig. 21.

Figure 21 – Programming Effort. Phase 2

Table 3 – Complexity metrics calculated for each method. Phase 1

Metric \ Approach	Variant 1 (New event introduction)	Varian 2 (Synchronous Queue)	Variant 3 (Causal Barrier)	Variant 4 (Container of Events)
Distinct operators	18	9	16	17
Distinct operands	55	4	29	40
Occurrences of operators	64	12	37	58
Occurrences of operands	86	8	62	62
Program Length	150	20	99	120
Halstead Vocabulary	72	13	45	57
Program Volume	925.49	74.01	543.69	699.95
Program Difficulty	14.33	9	17.1	13.18
Programming Effort	13262.27	666.09	9297.1	9225.34

Table 4 – Complexity metrics calculated for each method. Phase 2

Metric \ Approach	Variant 1 (New event introduction)	Varian 2 (Synchronous Queue)	Variant 3 (Causal Barrier)	Variant 4 (Container of Events)
Distinct operators	13	0	16	9
Distinct operands	48	0	30	19
Occurrences of operators	49	0	36	26
Occurrences of operands	77	0	63	29
Program Length	126	0	99	55
Halstead Vocabulary	61	0	46	28
Program Volume	747.27	0	546.83	264.4
Program Difficulty	10.43	0	16.8	6.87
Programming Effort	7794.03	0	9186.74	1816.43

DOI 10.15588/1607-3274-2024-3-11

Phase 3. For reasons of clarity, testing was conducted with the simplest possible command to minimize command validation time and aggregate state updates. Additionally, the time required for creating an aggregate was minimized through caching.

Table 5 contains average time metrics from several experiments repeated on three different computers. The chart in Fig. 22 represents the ratio of each approach metric to the fastest one in the category (which is Variant 1 – New event introduction) calculated by formula 30, that can roughly show the performance comparison of these solutions.

Phase 4. Table 6 contains the Integrated Performance-Complexity evaluation results for considered methods, calculated using formulas 31 and 32.

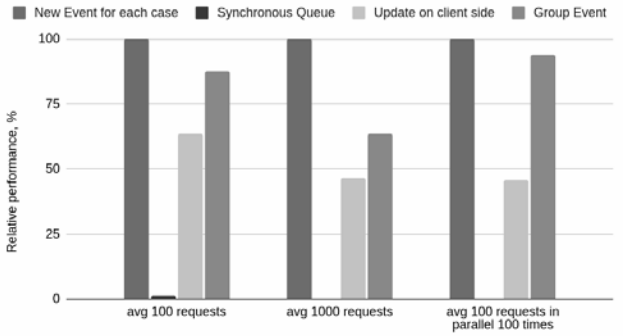

Figure 22 – Relative performance. Phase 3

Table 5 – Performance metrics calculated for each method. Phase 3

Approach \ Metric	avg 100 requests (ms)	avg 1000 requests (ms)	avg 100 requests in parallel 100 times (ms)
Variant 1 (New event introduction)	7	7	138
Variant 2 (Synchronous Queue)	453	4187	96117
Variant 3 (Causal Barrier)	11	15	301
Variant 4 (Container of Events)	8	11	147

Table 6 – Integrated Performance-Complexity evaluation. Phase 4

Metric \ Approach	Variant 1 (New event introduction)	Varian 2 (Synchronous Queue)	Variant 3 (Causal Barrier)	Variant 4 (Container of Events)
Average performance	100	0.62	52.05	81.67
Integration complexity	13262.27	666.09	9297.1	9225.34
Maintenance complexity	7794.03	0	9186.74	1816.43
Evaluation	9.49	1.86	5.63	14.79

6 DISCUSSION

According to the results presented in Table 3 the solution involving the synchronous queue (Variant 2) requires minimal Programming Effort (666.09).

The most challenging aspect is Causal Barrier solution (Variant 3). The volume of added code (543.69) is larger than that of adding for the synchronous queue variant (74.01), and the task itself proves considerably more complex (17.1 against 9) and demands more development and testing effort (9297.1 against 666.09).

The “New event introduction” (Variant 1) approach can be regarded as the simplest to implement, but due to the amount of routine work this approach is time-consuming. It can be used only for the systems with low degree of probability of casual events. It should also be noted that as the domain of the application gets more complex (i.e. new aggregates, functions, and causal events are introduced), this approach would lead to naming complexity causing the problems with maintainability of the application.

As it can be seen the Variant 4 (Container of Events) is in the second place in terms of effort (9225.34) and difficulty (13.18), following the synchronous queue solution (Variant 2. Difficulty: 9, Effort: 666.09). The effort invested in the Container of Events method (Variant 4. 9225.34) implementation is nearly identical to that of Causal Barrier (Variant 3. 9297.1). However, the “New event introduction” method requires more development effort due to the amount of routine work (Variant 1 13262.27). It also takes the third place in terms of volume (699.95), following the solutions with a queue (Variant 2. 74.01) and “Causal Barrier” (Variant 3. 543.69). The first phase of the experiment includes just a simple case for two causal events. In the case of more events, the volume of the “Causal Barrier” approach remains stable, while the volume for the “Container of Events” approach decreases.

In accordance with the results presented in Table 4, for the “Causal Barrier” approach (Variant 3), the program effort remained almost unchanged in comparison

DOI 10.15588/1607-3274-2024-3-11

to the effort spent on Phase 1 (9297.1 vs 9186.74). For the "New event introduction" approach (Variant 1), the difficulty decreased due to fewer changes but remained relatively high (14.33 vs 10.43). Meanwhile, for Variants 2 and 4, the program effort significantly decreased in comparison with Phase 1. It can be explained by the fact that the main part of the code changes is applied to the infrastructure and the implementation complexity is almost independent of the number of causal events. For example, the "Container of Events" method (Variant 4) only requires routine updates to event handlers, while the "Synchronous Queue" implementation (Variant 2) used in the experiment does not require any additional changes at all.

The best performance was demonstrated by the "New event introduction" method (Variant 1). This is because this solution does not introduce any new logic into the system's workings. However, as previously described, frequent use of this approach can lead to significant code duplication and the expansion of a list of narrowly focused events.

As expected, the solution with a synchronous queue (Variant 2) turned out to be the slowest. The performance drawbacks are especially evident when sending a large number of commands in parallel (average: 96 seconds).

The performance of the Container of Events method (Variant 4) takes the second place, trailing slightly behind Variant 1. Causal Barrier approach (Variant 3) ranks third. For tests with 100 and 1000 consecutive requests, it performs slightly slower. However, under parallel load, the difference becomes twofold. In the context of our experiment, this is explained by the inability to scale the client (browser); for clients with "server" type, such a test should yield better results.

Thus, the following conclusion can be drawn:

– If the assessment shows that the likelihood of critical causal events is low, it is better to use the "New event introduction" method, taking into account its drawbacks.

– If the assessment indicates that the likelihood of critical causal events is high, it is preferable to use either the "Synchronous Queue" method (if the system will not be under heavy load) or the "Container of Events" method.

– If predicting the likelihood of critical causal events is challenging and the system may evolve, according to the integrated comparative assessment calculations for specific systems and conditions, the "Container of Events" method (Variant 4) can be considered as the most favourable with the score of 14.79 against 9.49 for the "New event introduction" solution, which takes second place.

The use of the "Synchronous Queue" solution (Variant 2) shows poor performance, making it a situational approach that doesn't align with our specific use cases. When comparing "Container of Events" (Variant 4) with the others, the program effort of implementing this approach is lower than the complexity of "Causal Barrier" (Variant 3) and the "New event introduction" approach (Variant 1). Nevertheless, just plain adding of new events for each case (Variant 1) works slightly faster. Across performance-appropriate solutions, the "Container of Events" solution effectively addresses the issue with the lowest development effort for implementation and maintenance. It is worth noting it helps to avoid code duplication.

CONCLUSIONS

The scientific novelty. For the first time the method of estimation of the likelihood of causal events occurring within the systems based on CQRS and ES architecture and its formal description are suggested. The method is based on analysis of entities, their interconnection and the analysis of use cases connected to the entities and their relationships. The variant of precise prediction of the critical causal events occurrence based on the history of existed solutions has been also provided.

The "Container of Events" method is firstly proposed to solve the problem of critical causal-events for system based on CQRS with ES architecture. It effectively addresses the issue for most of the researched systems with the lowest development effort for implementation (HSSM Effort: 9225.34) and maintenance (HSSM Effort: 1816.43) across performance-appropriate solutions and without code duplication. The method of Integrated Performance-Complexity evaluation which helps to make complexity-performance comparative assessment more descriptive is firstly proposed. Evaluation based on this method are 14.79 for the "Container of Events" method against 9.49 for the "New event introduction" solution, which takes second place.

Commonly used synchronization methods such as Variant 1 (New event introduction), Variant 2 (Synchronous Queue), and Variant 3 (Causal Barrier) are formalized and assessed.

The practical significance of the obtained results is as follows. The formalized and assessed methods can be used for effective real information systems development. The strategy of experiment conducting applied to assess the complexity of the modification can be used in practice to resolve similar tasks (e.g. conducting similar experiments). The proposed indicators and methods can be used to determine effective conditions for the experiments connected with the complexity and performance evaluation.

The proposed solution which is provided to the systems based on CQRS with ES Architecture can also be applied to other systems in which the sequential delivery of events is not guaranteed.

ACKNOWLEDGEMENTS

The experiment was conducted on the DBB Software company's [43] proprietary platform, which provided the necessary infrastructure and tools for data collection and analysis. This platform offered essential capabilities for our research, ensuring the accuracy and reliability of our experimental results.

DOI 10.15588/1607-3274-2024-3-11

We also want to express our deep appreciation to Volodymyr Khandetsky, Head of Electronic Department, for his valuable comments and suggestions.

REFERENCES

1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004, 534 p. ISBN: 978-0321125217.
2. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2019, 464 p. ISBN: 978-0132350884.
3. Newman S. Building Microservices: Designing Fine-Grained Systems 2nd Edition. O'Reilly, 2021, 500 p. ISBN: 978-1492034025.
4. Michelson B. M. Event-Driven Architecture Overview, *Patricia Seybold Group and Elemental Links*. Boston, 2011, 9 p. DOI: 10.1571/bda2-2-06cc.
5. Taylor H., Yochem A., Phillips L. et al. Event-Driven Architecture: How SOA Enables the Real-Time Enterprise. Addison-Wesley Professional, 2009, 272 p. ISBN: 978-0321591388.
6. Neamtiu I., Dumitras T. Cloud software upgrades: Challenges and opportunities, *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems: status, 26–26 September 2011: proceedings*. Williamsburg: IEEE, 2011, pp. 1–10. ISBN: 978-1457706479.
7. Tarkoma S. Publish / Subscribe Systems: Design and Principles. Wiley, 2012, 352 p. ISBN: 978-1119951544.
8. Brandolini A. Introducing EventStorming [Electronic resource]. Access mode: https://leanpub.com/introducing_eventstorming.
9. Stopford B. Designing Event-Driven Systems. O'Reilly Media, 2018, 171 p. ISBN: 978-1492038221.
10. Garofolo E. Practical Microservices. Build Event-Driven Architectures with Event Sourcing and CQRS. Pragmatic Bookshelf, 2020, 292 p. ISBN: 978-1680507799.
11. Hoffman K. Building Microservices with ASP.NET Core / K. Hoffman. O'Reilly Media, 2017, 232 p. ISBN: 978-1491961735.
12. Young G. CQRS Documents by Greg Young [Electronic resource]. Access mode: https:// cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.
13. Young G. Event Centric: Finding Simplicity in Complex Systems. Addison-Wesley Professional, 2017, 560 p. ISBN: 978-0321768223.
14. Burckhardt S. Principles of Eventual Consistency (Foundations and Trends(r) in Programming Languages). Now Publishers, 2014, 170 p. ISBN: 978-1601988584.
15. Practical and focused guide for survival in post-CQRS world. Projections. [Electronic resource]. Access mode: http://cqrs.wikidot.com/doc:projection.
16. Comartin D., Young G. Answers your Event Sourcing questions! [Electronic resource]. Access mode: https://codeopinion.com/greg-young-answers-your-event-sourcing-questions.
17. Lloyd W., Freedman M. J., Kaminsky M. et al. Don't Settle for Eventual Consistency, *Communications of the ACM,* 2014, Vol. 57, Issue 5, pp. 61–68. DOI: 10.1145/2596624.
18. Lamport L. Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM,* 1978, Vol. 21, Issue 7, pp. 558–565. DOI: 10.1145/359545.359563.
19. Mostéfaoui A., Raynal M., Tredan G. On the Fly Estimation of the Processes that Are Alive in an Asynchronous Message-Passing System, *IEEE Transactions on Parallel and Distributed Systems,* 2009, Vol. 20, Issue 6. pp. 778–787. DOI: 10.1109/TPDS.2009.12.
20. Microsoft documentation. CQRS pattern. Implementation issues and considerations [Electronic resource]. Access mode: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs#implementation-issues-and-considerations.
21. Young G. Versioning in an Event Sourced System [Electronic resource]. Access mode: https://leanpub.com/esversioning.
22. Schwarz R., Mattern F. Detecting causal relationships in distributed computations: In search of the holy grail, *Distributed Computing*, 1994, Vol. 7, pp. 149–174. DOI: 10.1007/BF02277859.
23. Vernon V. Implementing Domain-Driven Design. Addison Wesley, 2013, 656 p. ISBN: 978-0321834577.
24. Hens P., Snoeck M., Poels G. et al. A Petri Net Formalization of a Publish-Subscribe Process System, *Social Science Research Network*, 2011. DOI: 10.2139/ssrn.1886198.
25. Farmer W. M. The seven virtues of simple type theory, *Journal of Applied Logic*, 2008, Vol. 6, Issue 3, pp. 267–286. DOI: 10.1016/j.jal.2007.11.001.
26. Muhl G., Fiege L., Pietzuch P. Distributed Event-Based Systems. New York, Springer-Verlag, 2006, 388 p. ISBN: 978-3540326519.
27. Baldoni R., Contenti M., Piergiovanni S. T. et al.Modelling Publish/Subscribe Communication Systems: Towards a Formal Approach, *Object-Oriented Real-Time Dependable Systems : 8th IEEE International Workshop, 15–17 January 2003 : proceedings*. Guadalajara, WORDS, 2003, pp. 304–311. DOI: 10.1109/WORDS.2003.1218097.
28. Araujo J. P., Arantes L., Duarte E. P. et al. VCube-PS: A causal broadcast topic-based publish/subscribe system, *Journal of Parallel and Distributed Computing*, 2019, Vol. 125, pp. 18–30. DOI: 10.1016/j.jpdc.2018.10.011.
29. Ohlbach H. J., Koehler J. Modal logics, description logics and arithmetic reasoning, *Artificial Intelligence,* 1999, Vol. 109, pp. 1–31. DOI: 10.1016/S0004-3702(99)00011-9.
30. Badra F. Case Adaptation with Modal Logic: The Modal Adaptation, *Case-Based Reasoning Research and Development: 22nd International Conference Reason, 29 September 2014 – 1 October 2014: proceedings.* Cork, ICCBR, 2014.
31. Fidge C. J. Timestamps in Message-Passing Systems that Preserve Partial Ordering, *Australian Computer Science Communications*, 1988, Vol. 10, No. 1, pp. 56–66.
32. Mattern F. Algorithms for distributed termination detection, 1987. DOI: 10.1007/BF01782776.
33. Singh A. Matrix Clock Synchronization in the Distributed Computing Environment, *International Journal of Computer Science and Information Technologies,* 2015, Vol. 6, Issue 4. pp. 3510–3513.
34. Guidec F., Launay P., Mahéo T. Causal and Δ-causal broadcast in opportunistic networks, *Future Generation Computer Systems*, 2021, Volume 118, Issue 1, pp. 142–156. DOI: 10.1016/j.future.2020.12.024.
35. Wan F., Singh M. P. Commitments and Causality for Multiagent Design, *2nd International Joint Conference on Autonomous Agents and Multiagent Systems, 14 – 18 July 2003: proceedings.* Melbourne: AAMAS, 2003, pp. 749–756. DOI: 10.1145/860575.860696.
36. Preguiça N. M., Bauqero C., Almeida P. S. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors, *ACM symposium on Principles of distributed computing*, 2012, pp. 335–336 DOI: 10.1145/2332432.2332497.
37. Zhou S., Cai W., Turner S. J. Critical causal order of events in distributed virtual environments, *ACM Transactions on Multimedia Computing, Communications and Applications,* 2007, Vol. 3. DOI: 10.1145/1236471.1236474.
38. Baldoni R., Prakash R., Raynal M. Efficient Delta-Causal Broadcasting, *International Journal of Computer Systems Science and Engineering,* 1998, Vol. 13, pp. 263–271. DOI: 10.3923/jas.2009.1711.1718.
39. Schiper A., Eggli J., Sandoz A. A New Algorithm to Implement Causal Ordering, *Distributed Algorithms : 3rd International*

DOI 10.15588/1607-3274-2024-3-11

Workshop, 26–28 September 1989: proceedings. Nice, pp. 219–232. DOI: 10.1007/3-540-51687-5_45.
40. Carzaniga A., Rosenblum D. S., Wolf A. L. Design and Evaluation of a Wide-Area Event Notification Service, *ACM Transactions on Computer Systems (TOCS),* 2001, Vol. 19, Issue 3, pp. 332–383. DOI: 10.1145/380749.380767.
41. Esposito C., Cotroneo D., Gokhale A. Reliable Publish/Subscribe Middleware for Time-sensitive Internet-scale Applications, *Distributed Event-Based Systems : the 3rd ACM International Conference, 6–9 July 2009 : proceedings*. Nashville, DEBS, 2009, pp. 1–12. DOI: 10.1145/1619258.1619280.
42. Microsoft. Saga distributed transactions pattern [Electronic resource]. Access mode: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.
43. DBB Software's official company site [Electronic resource]. Access mode: https://dbbsoftware.com/.
44. Berardi D., Calvanese D., Giacomo G. Reasoning on UML Class Diagrams, *Artificial Intelligence*, 2005, Vol. 168, Issue 1–2, pp. 70–118. DOI: 10.1016/j.artint.2005.05.003.
45. Calvanese D., Lenzerini M., Nardi D. A unified framework for class based representation formalisms, *Principles of Knowledge Representation and Reasoning : 4th International Conference, 24–27 May 1994 : proceedings*. San Francisco, pp. 109–120.
46. Danforth S., Tomlinson C. Type theories and object-oriented programming, *ACM Computing Surveys,* 1988, Vol. 20, Issue 1, pp. 29–72. DOI: 10.1145/62058.62060.
47. Zaman Q., Nadeem A., Sindhu M. A. Formalizing the use case model: A model-based approach, *PLoS ONE,* 2020, Vol. 15, Issue 4. DOI: 10.1371/journal.pone.0231534.
48. Kautz O., Rumpe B., Wachtmeister L. Semantic Differencing of Use Case Diagrams, *Journal of Object Technology*, 2022, Vol. 21, Issue 3, pp. 1–14. DOI: 10.5381/jot.2022.21.3.a5.
49. Rosenberg D., Stephens M. Use Case Driven Object Modeling with UML. Theory and Practice. Apress, 2013, 472 p. ISBN: 978-1430243052.
50. Genova G., Llorens J., Quintana V. Digging into use case relationships, *The Unified Modeling Language : 5th International Conference, 30 September – 4 October 2002 : proceedings*. Berlin, UML, 2002, pp. 115–127. DOI: 10.1007/3-540-45800-X_10.
51. Daan. The Impact of Duplicate Code [Electronic resource] / Daan. – Access mode: https://levelup.gitconnected.com/the-impact-of-duplicate-code-31c0bceab831.
52. How Event-Driven Architectures Benefit from Stream Processing [Electronic resource]. Access mode: https://pandio.com/event-streams-queues/
53. Seshadri P. Handling out of order events in a Event driven systems [Electronic resource]. Access mode: https://medium.com/@prabhu.seshadri/handling-out-of-order-events-in-a-event-driven-systems-93349bd20c26.
54. Lombardi A. WebSocket. Lightweight Client-Server Communications 1st Edition. O'Reilly, 2015, 144 p. ISBN: 978-1449369279.
55. Hruzin D. Link to GitHub repository with experiment [Electronic resource]. Access mode: https://github.com/dmitryhruzin/causal-event-experiment.
56. Mens T. Research trends in structural software complexity, *Computer Science, Engineering*, 2016. DOI: 10.48550/arXiv.1608.01533.
57. Bogdan St. Software development cost estimation methods and research trends, *Computer Science,* 2003, Vol. 5, pp. 67–86. DOI: 10.7494/csci.2003.5.1.3608.
58. Sarala S., Jabbar A. Information flow metrics and complexity measurement, *Computer Science and Information Technology : 3rd International Conference, 9–11 July 2010 : proceedings.* Chengdu, ICCSIT, 2010,Vol. 2. pp. 575–578. DOI: 10.1109/ICCSIT.2010.5563667.
59. Misra S. Measurement of Cognitive Functional Sizes of Software, *International Journal of Software Science and Computational Intelligence,* 2009,Vol. a, Issue 2, pp. 91–100. DOI: 10.4018/jssci.2009040106.
60. Beyer D., Häring P. A Formal Evaluation of DepDegree Based on Weyuker's Properties, *Program Comprehension : the 22nd International Conference, 2–3 June 2014 : proceedings*. Hyderabad, ICSE, 2014, pp. 258–261. DOI: 10.1145/2597008.2597794.
61. McCabe T.J. A Complexity Measure, *IEEE Transactions on Software Engineering*, 1976, Vol. SE-2, Issue 4, pp. 308–320. DOI: 10.1109/TSE.1976.233837.
62. Halstead M. H. Elements of Software Science. New York, Elsevier, 1977, 128 p. ISBN: 978-0444002051.
63. Sync-Queue Node.js package. GitHub [Electronic resource]. Access mode: https://github.com/tessel/sync-queue.
64. Amazon Simple Queue Service [Electronic resource]. Access mode: https://aws.amazon.com/sqs/.
65. Apache Kafka [Electronic resource]. Access mode: https://kafka.apache.org/.
66. Blest D. Theory & Methods: Rank Correlation – an Alternative Measure, *Australian & New Zealand Journal of Statistics*, 2000, Vol. 42, Issue 1, pp. 101–111. DOI: 10.1111/1467-842X.00110.

Received 18.06.2024.
Accepted 30.08.2024.

УДК 614.2+574/578+004.38

КРИТИЧНІ ПРИЧИННО-НАСЛІДКОВІ ПОДІЇ В СИСТЕМАХ ЗАСНОВАНИХ НА ОСНОВІ АРХІТЕКТУРИ CQRS З EVENT SOURCING

Литвинов О. А. – канд. техн. наук, доцент кафедри електронних обчислювальних машин Дніпровського національного університету імені Олеся Гончара, Дніпро, Україна.

Грузін Д. Л. – аспірант кафедри електронних обчислювальних машин Дніпровського національного університету імені Олеся Гончара, Дніпро, Україна.

АНОТАЦІЯ

Актуальність. У статті розглядається проблема асинхронності причинно-наслідкових подій, що виникає в сервісно-орієнтованих інформаційних системах, які не гарантують доставку подій у порядку їх публікації. Це може призвести до помилок, які виникають випадково, як правило нерегулярно, у системі, яка протягом основного часу функціонує без збоїв.

Мета роботи. Метою роботи є порівняння та оцінка кількох існуючих підходів та пропонування нового підходу до вирішення проблеми синхронізації причинно-наслідкових подій у системах, які побудовані з застосуванням архітектури Command Query Responsibility Segregation (CQRS) з Event Sourcing (ES).

Методи. По-перше, пропонується метод оцінки ймовірності виникнення причинно-наслідкових подій у системах, як основа для вибору рішення. Так, на основі результатів аналізу кількох проектів, побудованих з застосуванням архітектури CQRS з ES, показано, що ймовірність критичних причинно-наслідкових подій залежить від взаємозв'язків між сутностями та юз-кейсів, пов'язаних із сутностями. По-друге, у цій роботі пропонується метод "Container of events", який представляє варіацію події з

DOI 10.15588/1607-3274-2024-3-11

повною історією причинно-наслідкових зв'язків, адаптовану до потреб систем побудованих з застосуванням архітектури CQRS з ES. Також обговорено варіанти його практичного впровадження. Крім того, були формалізовані та оцінені різні рішення, такі як синхронні черги подій та варіація методу "Causal Barrier". По-третє, представлені методи, були описані та оцінені за критеріями продуктивності та складності модифікації. Для отримання порівняльної оцінки складності та продуктивності була вперше запропонована інтегрована формула оцінки.

Результати. Результати оцінки показують, що найефективнішим рішенням проблеми є використання методу "Container of events". Для впровадження рішення необхідно внести зміни до підсистеми доставки подій та інфраструктури обробки подій.

Висновки. Робота зосереджена на вирішенні проблеми критичних причинно-наслідкових подій для систем, побудованих з застосуванням архітектури CQRS з ES. Запропоновано метод оцінки ймовірності виникнення критичних причинно-наслідкових подій, а також формалізовано та оцінено різні рішення цієї проблеми. Було запропоновано найефективніше рішення на основі методу "Container of events".

КЛЮЧОВІ СЛОВА: Сервісно-Орієнтована Архітектура, Архітектура, заснована на подіях, Event sourcing, Синхронізація подій, Проектування на основі домену.

ЛІТЕРАТУРА

1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans. – Addison-Wesley Professional, 2004. – 534 p. ISBN: 978-0321125217.
2. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship / R. C. Martin. – Prentice Hall, 2019. – 464 p. ISBN: 978-0132350884.
3. Newman S. Building Microservices: Designing Fine-Grained Systems 2nd Edition / S. Newman. – O'Reilly, 2021. – 500 p. ISBN: 978-1492034025.
4. Michelson B. M. Event-Driven Architecture Overview / B. M. Michelson // Patricia Seybold Group and Elemental Links. – Boston, 2011. – 9 p. DOI: 10.1571/bda2-2-06cc.
5. Event-Driven Architecture: How SOA Enables the Real-Time Enterprise / [H. Taylor, A. Yochem, L. Phillips et al.]. – Addison-Wesley Professional, 2009. – 272 p. ISBN: 978-0321591388.
6. Neamtiu I. Cloud software upgrades: Challenges and opportunities / I. Neamtiu, T. Dumitras // 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems: status, 26–26 September 2011: proceedings. – Williamsburg: IEEE, 2011. – P. 1–10. ISBN: 978-1457706479.
7. Tarkoma S. Publish / Subscribe Systems: Design and Principles / S. Tarkoma. – Wiley, 2012. – 352 p. ISBN: 978-1119951544.
8. Brandolini A. Introducing EventStorming [Electronic resource] / A. Brandolini. – Access mode: https://leanpub.com/introducing_eventstorming.
9. Stopford B. Designing Event-Driven Systems / B. Stopford. – O'Reilly Media, 2018. – 171 p. ISBN: 978-1492038221.
10. Garofolo E. Practical Microservices. Build Event-Driven Architectures with Event Sourcing and CQRS / E. Garofolo. – Pragmatic Bookshelf, 2020. – 292 p. ISBN: 978-1680507799.
11. Hoffman K. Building Microservices with ASP.NET Core / K. Hoffman. – O'Reilly Media, 2017. – 232 p. ISBN: 978-1491961735.
12. Young G. CQRS Documents by Greg Young [Electronic resource] / G. Young. – Access mode: https:// cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.
13. Young G. Event Centric: Finding Simplicity in Complex Systems / G. Young. – Addison-Wesley Professional, 2017. – 560 p. ISBN: 978-0321768223.
14. Burckhardt S. Principles of Eventual Consistency (Foundations and Trends(r) in Programming Languages) / S. Burckhardt. – Now Publishers, 2014. – 170 p. ISBN: 978-1601988584.
15. Practical and focused guide for survival in post-CQRS world. Projections. [Electronic resource]. – Access mode: http://cqrs.wikidot.com/doc:projection.
16. Comartin D. Answers your Event Sourcing questions! [Electronic resource] / D. Comartin, G. Young. – Access mode: https://codeopinion.com/greg-young-answers-your-event-sourcing-questions.
17. Don't Settle for Eventual Consistency / [W. Lloyd, M. J. Freedman, M. Kaminsky et al.] // Communications of the ACM. – 2014. – Vol. 57, Issue 5. – P. 61–68. DOI: 10.1145/2596624.
18. Lamport L. Time, clocks, and the ordering of events in a distributed system / L. Lamport // Communications of the ACM. – 1978. – Vol. 21, Issue 7. – P. 558–565. DOI: 10.1145/359545.359563.
19. Mostéfaoui A. On the Fly Estimation of the Processes that Are Alive in an Asynchronous Message-Passing System / A. Mostéfaoui, M. Raynal, G. Tredan // IEEE Transactions on Parallel and Distributed Systems. – 2009. – Vol. 20, Issue 6. – P. 778–787. DOI: 10.1109/TPDS.2009.12.
20. Microsoft documentation. CQRS pattern. Implementation issues and considerations [Electronic resource]. – Access mode: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs#implementation-issues-and-considerations.
21. Young G. Versioning in an Event Sourced System [Electronic resource] / G. Young. – Access mode: https://leanpub.com/esversioning.
22. Schwarz R. Detecting causal relationships in distributed computations: In search of the holy grail / R. Schwarz, F. Mattern // Distributed Computing. – 1994. – Vol. 7. – P. 149–174. DOI: 10.1007/BF02277859.
23. Vernon V. Implementing Domain-Driven Design / V. Vernon. – Addison Wesley, 2013. – 656 p. ISBN: 978-0321834577.
24. A Petri Net Formalization of a Publish-Subscribe Process System / [P. Hens, M. Snoeck, G. Poels et al.] // Social Science Research Network. – 2011. DOI: 10.2139/ssrn.1886198.
25. Farmer W. M. The seven virtues of simple type theory / W. M. Farmer // Journal of Applied Logic. – 2008. – Vol. 6, Issue 3. – P. 267–286. DOI: 10.1016/j.jal.2007.11.001.
26. Muhl G. Distributed Event-Based Systems / G. Muhl, L. Fiege, P. Pietzuch. – New York : Springer-Verlag, 2006. – 388 p. ISBN: 978-3540326519.
27. Modelling Publish/Subscribe Communication Systems: Towards a Formal Approach / [R. Baldoni, M. Contenti, S. T. Piergiovanni et al.] // Object-Oriented Real-Time Dependable Systems : 8th IEEE International Workshop, 15–17 January 2003 : proceedings. – Guadalajara: WORDS, 2003. – P. 304–311. DOI: 10.1109/WORDS.2003.1218097.
28. VCube-PS: A causal broadcast topic-based publish/subscribe system / [J. P. Araujo, L. Arantes, E. P. Duarte et al.] // Journal of Parallel and Distributed Computing. – 2019. – Vol. 125. – P. 18–30. DOI: 10.1016/j.jpdc.2018.10.011.
29. Ohlbach H. J. Modal logics, description logics and arithmetic reasoning / H. J. Ohlbach, J. Koehler // Artificial Intelligence. – 1999. – Vol. 109. – P. 1–31. DOI: 10.1016/S0004-3702(99)00011-9.
30. Badra F. Case Adaptation with Modal Logic: The Modal Adaptation / F. Badra // Case-Based Reasoning Research and Development: 22nd International Conference Reason, 29 September 2014 – 1 October 2014: proceedings. – Cork : ICCBR, 2014.

DOI 10.15588/1607-3274-2024-3-11

31. Fidge C. J. Timestamps in Message-Passing Systems that Preserve Partial Ordering / C. J. Fidge // Australian Computer Science Communications. – 1988. – Vol. 10, No. 1. – P. 56–66.
32. Mattern F. Algorithms for distributed termination detection / F. Mattern. – 1987. DOI: 10.1007/BF01782776.
33. Singh A. Matrix Clock Synchronization in the Distributed Computing Environment / A. Singh // International Journal of Computer Science and Information Technologies. – 2015. – Vol. 6, Issue 4. – P. 3510–3513.
34. Guidec F. Causal and Δ-causal broadcast in opportunistic networks / F. Guidec, P. Launay, T. Mahéo // Future Generation Computer Systems. – 2021. – Volume 118, Issue 1 – P. 142–156. DOI: 10.1016/j.future.2020.12.024.
35. Wan F. Commitments and Causality for Multiagent Design / F. Wan, M. P. Singh // 2nd International Joint Conference on Autonomous Agents and Multiagent Systems, 14–18 July 2003: proceedings. – Melbourne : AAMAS, 2003. – P. 749–756. DOI: 10.1145/860575.860696.
36. Preguiça N. M. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors / N. M. Preguiça, C. Bauqero, P. S. Almeida // ACM symposium on Principles of distributed computing. – 2012. – P. 335–336 DOI: 10.1145/2332432.2332497.
37. Zhou S. Critical causal order of events in distributed virtual environments / S. Zhou, W. Cai, S. J. Turner // ACM Transactions on Multimedia Computing, Communications and Applications. – 2007. – Vol. 3. DOI: 10.1145/1236471.1236474.
38. Baldoni R. Efficient Delta-Causal Broadcasting / R. Baldoni, R. Prakash, M. Raynal // International Journal of Computer Systems Science and Engineering. – 1998. – Vol. 13. – P. 263–271. DOI: 10.3923/jas.2009.1711.1718.
39. Schiper A. A New Algorithm to Implement Causal Ordering / A. Schiper, J. Eggli, A. Sandoz // Distributed Algorithms : 3rd International Workshop, 26–28 September 1989: proceedings. – Nice. – P. 219–232. DOI: 10.1007/3-540-51687-5_45.
40. Carzaniga A. Design and Evaluation of a Wide-Area Event Notification Service / A. Carzaniga, D. S. Rosenblum, A. L. Wolf // ACM Transactions on Computer Systems (TOCS). – 2001. – Vol. 19, Issue 3. – P. 332–383. DOI: 10.1145/380749.380767.
41. Esposito C. Reliable Publish/Subscribe Middleware for Time-sensitive Internet-scale Applications / C. Esposito, D. Cotroneo, A. Gokhale // Distributed Event-Based Systems : the 3rd ACM International Conference, 6–9 July 2009 : proceedings. – Nashville: DEBS, 2009. – P. 1–12. DOI: 10.1145/1619258.1619280.
42. Microsoft. Saga distributed transactions pattern [Electronic resource]. – Access mode: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.
43. DBB Software's official company site [Electronic resource]. – Access mode: https://dbbsoftware.com/.
44. Berardi D. Reasoning on UML Class Diagrams / D. Berardi, D. Calvanese, G. Giacomo // Artificial Intelligence. – 2005. – Vol. 168, Issue 1–2. – P. 70–118. DOI: 10.1016/j.artint.2005.05.003.
45. Calvanese D. A unified framework for class based representation formalisms / D. Calvanese, M. Lenzerini, D. Nardi // Principles of Knowledge Representation and Reasoning : 4th International Conference, 24–27 May 1994 : proceedings. – San Francisco. – P. 109–120.
46. Danforth. S. Type theories and object-oriented programming / S. Danforth, C. Tomlinson // ACM Computing Surveys. – 1988. – Vol. 20, Issue 1. – P. 29–72. DOI: 10.1145/62058.62060.
47. Zaman Q. Formalizing the use case model: A model-based approach / Q. Zaman, A. Nadeem, M. A. Sindhu // PLoS ONE. – 2020. – Vol. 15, Issue 4. DOI: 10.1371/journal.pone.0231534.
48. Kautz O. Semantic Differencing of Use Case Diagrams / O. Kautz, B. Rumpe, L. Wachtmeister // Journal of Object Technology. – 2022. – Vol. 21, Issue 3. – P. 1–14. DOI: 10.5381/jot.2022.21.3.a5.
49. Rosenberg D. Use Case Driven Object Modeling with UML. Theory and Practice / D. Rosenberg, M. Stephens. – Apress, 2013. – 472 p. ISBN: 978-1430243052.
50. Genova G. Digging into use case relationships / G. Genova, J. Llorens, V. Quintana // The Unified Modeling Language : 5th International Conference, 30 September – 4 October 2002 : proceedings. – Berlin : UML, 2002. – P. 115–127. DOI: 10.1007/3-540-45800-X_10.
51. Daan. The Impact of Duplicate Code [Electronic resource] / Daan. – Access mode: https://levelup.gitconnected.com/the-impact-of-duplicate-code-31c0bceab831.
52. How Event-Driven Architectures Benefit from Stream Processing [Electronic resource]. – Access mode: https://pandio.com/event-streams-queues/
53. Seshadri P. Handling out of order events in a Event driven systems [Electronic resource] / P. Seshadri. – Access mode: https://medium.com/@prabhu.seshadri/handling-out-of-order-events-in-a-event-driven-systems-93349bd20c26.
54. Lombardi A. WebSocket. Lightweight Client-Server Communications 1st Edition / A. Lombardi. – O'Reilly, 2015. – 144 p. ISBN: 978-1449369279.
55. Hruzin D. Link to GitHub repository with experiment [Electronic resource] / D. Hruzin. – Access mode: https://github.com/dmitryhruzin/causal-event-experiment.
56. Mens T. Research trends in structural software complexity / T. Mens // Computer Science, Engineering. – 2016. DOI: 10.48550/arXiv.1608.01533.
57. Bogdan St. Software development cost estimation methods and research trends / St. Bogdan // Computer Science. – 2003. – Vol. 5. – P 67–86. DOI: 10.7494/csci.2003.5.1.3608.
58. Sarala S. Information flow metrics and complexity measurement / S. Sarala, A. Jabbar // Computer Science and Information Technology : 3rd International Conference, 9–11 July 2010 : proceedings. – Chengdu: ICCSIT, 2010. – Vol. 2. – P. 575–578. DOI: 10.1109/ICCSIT.2010.5563667.
59. Misra S. Measurement of Cognitive Functional Sizes of Software / S. Misra // International Journal of Software Science and Computational Intelligence. – 2009. – Vol. a, Issue 2. – P: 91–100. DOI: 10.4018/jssci.2009040106.
60. Beyer D. A Formal Evaluation of DepDegree Based on Weyuker's Properties / D. Beyer, P. Häring // Program Comprehension : the 22nd International Conference, 2–3 June 2014 : proceedings. – Hyderabad: ICSE, 2014. – P. 258–261. DOI: 10.1145/2597008.2597794.
61. McCabe T. J. A Complexity Measure / T. J. McCabe // IEEE Transactions on Software Engineering. – 1976. –Vol. SE-2, Issue 4. – P. 308–320. DOI: 10.1109/TSE.1976.233837.
62. Halstead M. H. Elements of Software Science / M. H. Halstead. – New York : Elsevier, 1977. – 128 p. ISBN: 978-0444002051.
63. Sync-Queue Node.js package. GitHub [Electronic resource]. – Access mode: https://github.com/tessel/sync-queue.
64. Amazon Simple Queue Service [Electronic resource]. – Access mode: https://aws.amazon.com/sqs/.
65. Apache Kafka [Electronic resource]. – Access mode: https://kafka.apache.org/.
66. Blest D. Theory & Methods: Rank Correlation — an Alternative Measure / D. Blest // Australian & New Zealand Journal of Statistics. – 2000. – Vol. 42, Issue 1. – P. 101–111. DOI: 10.1111/1467-842X.00110.

DOI 10.15588/1607-3274-2024-3-11