http://dx.doi.org/10.35596/1729-7648-2026-24-1-68-74

УДК 004.42; 004.5

АЛГОРИТМ ВЗАИМОДЕЙСТВИЯ КОМПОНЕНТОВ НА СТОРОНЕ КЛИЕНТА

А. А. СТРУКОВА, Л. Ю. ШИЛИН

Белорусский государственный университет информатики и радиоэлектроники (Минск, Республика Беларусь)

Аннотация. В статье рассматривается проблема организации эффективного взаимодействия компонентов в клиентских веб-приложениях, построенных на основе компонентных архитектур. Показано, что при увеличении масштаба интерфейсных систем традиционные подходы к обмену данными между компонентами приводят к избыточным вычислениям, усложнению поддержки и снижению производительности. На примере фреймворков проанализированы существующие механизмы синхронизации состояния и обновления пользовательского интерфейса, выявлены их ограничения, связанные с глобальным отслеживанием изменений и отсутствием встроенной поддержки двусторонней передачи данных между компонентами. Предложена двухсвязная модель взаимодействия компонентов (BCLM), основанная на прямом обмене релевантными изменениями состояния между логически связанными компонентами. Формализован механизм синхронизации через отображения публикации и применение данных, обеспечивающий адаптивное обновление интерфейса без участия централизованных хранилищ. Показано, что предложенный подход позволяет уменьшить количество перерендерингов и упростить архитектуру приложения при сохранении реактивности.

Ключевые слова: веб-приложения, состояние компонента, синхронизация данных, реактивность, Angular, React, виртуальный DOM, модель BCLM, оптимизация производительности.

Конфликт интересов. Авторы заявляют об отсутствии конфликта интересов.

Для цитирования. Струкова, А. А. Алгоритм взаимодействия компонентов на стороне клиента / А. А. Струкова, Л. Ю. Шилин // Доклады БГУИР. 2026. Т. 24, № 1. С. 68–74. http://dx.doi.org/10.35596/1729-7648-2026-24-1-68-74.

ALGORITHM OF COMPONENTS INTERACTION ON THE CLIENT SIDE

ALINA STRUKOVA, LEONID SHILIN

Belarusian State University of Informatics and Radioelectronics (Minsk, Republic of Belarus)

Abstract. The article discusses the problem of organizing effective component interaction in client web applications based on component architectures. It is shown that as the scale of interface systems increases, traditional approaches to data exchange between components lead to redundant calculations, more complicated support, and lower performance. Using the example of frameworks, the existing mechanisms for synchronizing the state and updating the user interface are analyzed, their limitations related to global change tracking and the lack of built-in support for two-way data transfer between components are identified. A Bidirectional Component Linking Model (BCLM) is proposed, based on the direct exchange of relevant state changes between logically connected components. A synchronization mechanism has been formalized through the display of publication and application of data, providing adaptive interface updates without the involvement of centralized repositories. It is shown that the proposed approach makes it possible to reduce the number of re-renderings and simplify the application architecture while maintaining reactivity.

Keywords: web applications, component state, data synchronization, reactivity, Angular, React, virtual DOM, BCLM model, performance optimization.

Conflict of interests. The authors declare that there is no conflict of interests.

For citation. Strukova A., Shilin L. (2026) Algorithm of Components Interaction on the Client Side. *Doklady BGUIR*. 24 (1), 68–74. http://dx.doi.org/10.35596/1729-7648-2026-24-1-68-74 (in Russian).

Введение

В последние годы наблюдается устойчивый рост популярности компонентных архитектур при создании клиентских веб-приложений. Данный подход предполагает разделение пользовательского интерфейса на независимые логические модули – компоненты, каждый из которых инкапсулирует собственное состояние, логику и представление. Это значительно упрощает разработку, сопровождение и тестирование программных систем, поскольку позволяет повторно использовать элементы интерфейса и локализовать изменения в пределах отдельных блоков.

Существующие фреймворки, наиболее широко применяемые в индустрии (Angular и React), предлагают различные модели обмена данными и обновления интерфейса. Несмотря на их практическую эффективность, в условиях роста масштаба приложения проявляются архитектурные ограничения, связанные с избыточными вычислениями и усложнением синхронизации состояний.

В связи с этим актуальной задачей является разработка подхода к взаимодействию компонентов, обеспечивающего согласованность данных и оптимальное использование вычислительных ресурсов. Перспективным направлением выступает применение моделей, позволяющих организовать прямой обмен состояниями между связанными компонентами без привлечения глобальных механизмов обновления. Одно из перспективных решений – двухсвязная модель взаимодействия компонентов (Bidirectional Component Linking Model, BCLM), обеспечивающая прямую реактивную синхронизацию только между логически зависимыми частями интерфейса, без вмешательства глобальных механизмов обновления.

Архитектурные принципы

Angular реализует архитектурную модель MVVM (Model – View – View Model), в которой каждый компонент C_i содержит локальное состояние S_i, отображаемое в шаблон V_i. Связывание данных может быть представлено как отображение

$$V_i = R(S_i), \tag{1}$$

где R – функция рендеринга.

Передача данных между компонентами осуществляется тремя основными способами. Первый представляет собой входные параметры (input binding), которые направляют поток данных сверху вниз. Связь осуществляется через декоратор @Input(). Это означает, что дочерний компонент получает значения, определяемые родителем. Такое взаимодействие компонентов можно представить формулой

$$S_{child}(t) = F\left(S_{parent}(t)\right). \tag{2}$$

Выходные параметры (output binding) – поток снизу вверх – являются вторым способом передачи данных между компонентами. Взаимодействие осуществляется через декоратор @Output(), что позволяет дочернему компоненту сообщать родителю о том, что его состояние изменилось (например, пользователь нажал кнопку). Этот способ описывается формулой

$$E_{parent}(t+1) = g\left(E_{child}(t)\right). \tag{3}$$

Одним из способов взаимодействия компонентов являются реактивные потоки RxJS, описываемые через подписки. Фактически это означает, что компонент обновляется, когда происходит определенное событие. Взаимодействие можно описать следующим выражением:

$$S_i(t+1) = S_i(t) + \delta S. \tag{4}$$

Каждое изменение состояния инициирует механизм Change Detection, проходящий по дереву компонентов и проверяющий, произошло ли изменение значений, используемых в шаблоне.

Change Detection представляет собой процесс выявления изменений в состоянии приложения и последующего обновления элементов пользовательского интерфейса, зависимых от этих данных.

Angular использует стратегию итеративного обхода дерева компонентов. При возникновении любого события, способного изменить состояние (обработчик событий, завершение HTTP-запроса, таймер и др.), Angular инициирует глобальный цикл Change Detection. В ходе этого цикла фреймворк последовательно проверяет все компоненты, сравнивая их текущее состояние с предшествующим. Это обеспечивает предсказуемость поведения, однако в крупных приложениях вызывает значительные накладные расходы при рендеринге и синхронизации состояний.

React использует концепцию однонаправленного потока данных, где родительский компонент передает значения свойств дочерним. Родитель передает свойства дочерним компонентам, а те реагируют на изменения состояния с помощью хуков (useState, useEffect, useMemo). Это взаимодействие можно описать формулой

$$Props_{child}(t) = h\left(State_{parent}(t)\right). \tag{5}$$

Изменение состояния через хуки вызывает пересчет виртуального дерева элементов – Virtual DOM. Текущее дерево интерфейса и новое состояние дерева элементов после изменения можно обозначить как DOM_{old} и DOM_{new} соответственно. React вычисляет разницу как разность между ними

$$\Delta DOM = DOM_{new} - DOM_{old}. \tag{6}$$

Это делает React более производительным, особенно при частых обновлениях интерфейса. Для сложных взаимодействий между множеством компонентов часто требуются дополнительные библиотеки для глобального управления состоянием (Redux, MobX, Zustand), что увеличивает когнитивную сложность системы.

Проблемы существующих решений

Сопоставление архитектурных принципов Angular и React позволяет выявить ряд характерных ограничений, оказывающих влияние на производительность и масштабируемость крупных клиентских приложений. Оба фреймворка эффективно решают задачу разделения интерфейса на независимые компоненты, но их механизмы синхронизации данных и обновления пользовательского интерфейса демонстрируют принципиальные различия.

Во-первых, в Angular основная проблема связана с высокой нагрузкой на систему обнаружения изменений. Поскольку механизм Change Detection инициируется при любом событии, потенциально влияющем на состояние данных, в крупных приложениях происходит повторяющийся обход обширного дерева компонентов. Даже при отсутствии фактических изменений во многих узлах компоненты все равно подвергаются проверке, что приводит к значительным вычислительным накладным расходам и увеличивает время отклика интерфейса. Таким образом, модель гарантированной синхронизации состояния и представления достигается ценой снижения эффективности при масштабировании.

Во-вторых, в React рендеринг оптимизирован благодаря сравнению виртуальных представлений интерфейса, однако отсутствует встроенный механизм двухсторонней синхронизации данных между компонентами. Передача данных осуществляется строго сверху вниз, что усложняет реализацию взаимодействия между компонентами, находящимися на разных уровнях иерархии. Для решения этой проблемы обычно применяются внешние хранилища состояния (Redux, MobX, Zustand), что увеличивает архитектурную сложность, расширяет количество связей в системе и требует дополнительных усилий по поддержке.

В-третьих, для обеих архитектур характерна проблема избыточной передачи данных по цепочкам компонентов. Даже если изменение состояния затрагивает лишь локальный участок интерфейса, данные часто приходится прокидывать через промежуточные компоненты, которые сами в изменении не участвуют. Это не только увеличивает связность системы, но и усложняет сопровождение и рефакторинг кода, поскольку локальные обновления могут непредсказуемо влиять на поведение различных частей приложения.

Указанные ограничения показывают, что в современных фреймворках отсутствует универсальный механизм, который одновременно обеспечивал бы производительное обновление интерфейса, сохранение реактивности данных и упрощенное взаимодействие между зависимыми компонентами без посредников. Следовательно, актуальной представляется задача разработки алгоритма взаимодействия компонентов, который бы минимизировал число перерендерингов, поддерживал прямую синхронизацию зависимых представлений и устранял необходимость избыточной передачи данных через цепочки компонентов. Такой подход позволил бы объединить сильные стороны существующих моделей, одновременно снижая их архитектурные и вычислительные издержки.

Двухсвязная модель взаимодействия компонентов

В рамках данной работы предлагается двухсвязная модель взаимодействия компонентов BCLM, на основе которой формализуется алгоритм синхронизации состояний между компонентами. В отличие от классического однонаправленного потока данных, в рамках BCLM каждый компонент не только обладает собственным локальным состоянием, но и способен публиковать изменения своих данных, а также подписываться на изменения состояния других компонентов без участия промежуточных элементов и глобального хранилища. Это обеспечивает прямую семантическую зависимость между компонентами, расположенными как на соседних, так и на удаленных уровнях иерархии.

Пусть C_i и C_j – два компонента системы с локальными состояниями $S_i(t)$ и $S_j(t)$, определяемыми в момент времени t. Тогда отношение двунаправленной связи между ними представляется следующим образом:

$$L_{i,j} = \left(C_i \leftrightarrow C_j\right). \tag{7}$$

Данное отношение не накладывает иерархических ограничений: компоненты могут находиться в любой конфигурации структуры представления. Пусть в компоненте C_i произошло изменение состояния

$$S_i\left(t+1\right) = S_i\left(t\right) + \delta S_i\left(t\right). \tag{8}$$

Не каждое изменение состояния является значимым для других компонентов, поэтому вводится отображение $h()$, выделяющее релевантную часть состояния:

$$\tau_i\left(t\right) = h\left(\delta S_i\left(t\right)\right), \tag{9}$$

где $\tau_{i,j}(t)$ – публикуемое изменение, предназначенное для доставки к подписчику C_j.

Соответственно, обновление состояния подписывающегося компонента C_j описывается функцией применения полученных данных

$$S_j\left(t+1\right) = f\left(S_j\left(t\right),\ \tau_{i,j}\left(t\right)\right). \tag{10}$$

Таким образом, итерация синхронизации между компонентами задается следующим отображением:

$$F_{i\to j} : S_j\left(t\right) \mapsto S_j\left(t+1\right). \tag{11}$$

Для того чтобы обновления состояний не приводили к бесконечным циклам (например, при $C_i \leftrightarrow C_j$), требуется условие сходимости синхронизационного процесса. Введем оператор глобального состояния для системы из n компонентов

$$S\left(t\right) = \left(S_1\left(t\right),\ S_2\left(t\right),\ ...,\ S_n\left(t\right)\right). \tag{12}$$

Тогда процесс двунаправленных синхронизаций представляет собой итеративное отображение

$$S\left(t+1\right) = \Phi\left(S\left(t\right)\right). \tag{13}$$

Чтобы модель была корректной, отображение Ф должно удовлетворять нескольким критериям. Отображение должно быть стабильным, чтобы при фиксированных входных данных не изменять состояние бесконечно. Изменение одного состояния не должно инициировать обновление тех компонентов, которые не находятся в зависимости, поэтому Ф должно быть локализованным. Также отображение должно быть монотонным относительно применения обновлений. Эти условия выполняются, если:

– h выделяет только семантически релевантные изменения;

– f является поглощающим оператором, т. е. не генерирует новое состояние, если изменений нет;

– циклы синхронизаций разрешаются приоритетным порядком применений (например, через очередь событий).

В общем случае взаимодействие компонентов на стороне клиента с помощью двухсвязного алгоритма можно представить схемой на рис. 1.

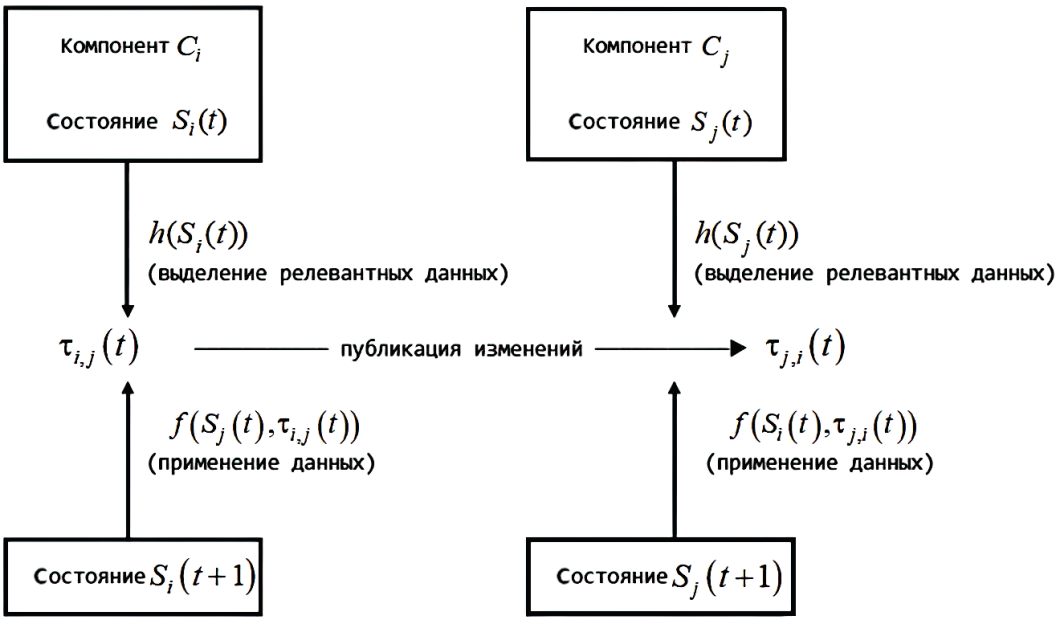

Рис. 1. Взаимодействие компонентов
Fig. 1. Components interaction

Предложенная модель BCLM, в которой распространение изменений ограничено только теми элементами интерфейса, действительно находящимися в логической зависимости. Это позволяет отказаться от глобального обхода дерева компонентов (характерного для Angular) и от централизованных хранилищ состояния (широко используемых в экосистеме React).

Процесс обновления пользовательского интерфейса становится локализованным: изменение состояния компонента приводит к перерасчету и перерисовке только тех элементов, которые непосредственно подписаны на данное состояние. Таким образом, достигается снижение количества избыточных вычислений и сокращается объем обновлений в DOM-дереве.

Важным следствием двунаправленности модели является возможность естественного распространения изменений в обе стороны зависимости, когда каждый компонент одновременно выступает и источником, и приемником данных. Это позволяет отказаться от каскадной передачи свойств через цепочки промежуточных компонентов, что существенно уменьшает связность кода и снижает когнитивную нагрузку.

С точки зрения временной сложности предложенный подход переводит операцию обновления состояния из класса $O(|T|)$ (где $|T|$ – размер дерева компонентов) в $O(|L|)$ (где $|L|$ – число логических связей между компонентами). Первый класс характерен для механизмов глобального Change Detection. Поскольку на практике $|L| << |T|$, особенно в масштабируемых приложениях, это приводит к существенному улучшению производительности интерфейса.

Кроме того, модель сохраняет реактивность обновлений: состояние интерфейса остается согласованным при любых последовательностях изменений, так как каждое обновление транслируется по строго определенным зависимостям. Это гарантирует детерминированность поведения системы, что является ключевым условием для обеспечения корректности работы многокомпонентных SPA-приложений.

Заключение

1. Проведенный анализ современных подходов к организации взаимодействия компонентов в клиентских веб-приложениях показал, что существующие архитектурные решения, такие как однонаправленный поток данных и централизованное управление состоянием, обладают как существенными преимуществами, так и рядом ограничений, проявляющихся при масштабировании сложных интерфейсных систем. В рамках таких моделей наблюдается либо избыточная вычислительная нагрузка, обусловленная глобальными механизмами отслеживания изменений, либо повышенная когнитивная сложность разработки, возникающая вследствие необходимости явного управления потоками данных между компонентами.

2. Предложенная двухсвязная модель взаимодействия компонентов (Bidirectional Component Linking Model, BCLM) обеспечивает более гибкий и локализованный механизм синхронизации состояний. За счет прямого обмена релевантными изменениями между логически зависимыми компонентами достигается уменьшение числа избыточных обновлений, сокращается глубина цепочек передачи данных и обеспечивается адаптивность интерфейсной системы. Формализация процесса синхронизации в виде отображений $h()$ и $f()$ позволяет четко определить границы релевантности изменений и способы их применения на стороне подписчика, а введенные условия сходимости исключают возможность формирования бесконечных циклов обновления.

3. Предлагаемая модель объединяет преимущества реактивного обновления интерфейса и структурированного управления состоянием, минимизируя вычислительные затраты и сохраняя ясность архитектуры. Дальнейшее направление работы включает разработку экспериментального прототипа, проведение сравнительного анализа производительности относительно существующих фреймворков и исследование оптимальных стратегий выбора функций преобразования состояний для различных типов приложений.

Список литературы

1. Herman, D. Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript / D. Herman. Boston: Addison-Wesley, 2013
2. Флэнаган, Д. JavaScript. Подробное руководство / Д. Флэнаган. СПб.: Символ-Плюс, 2021.
3. Bampakos, A. Learning Angular: A No-Nonsense Guide to Building Web Applications with Angular 15 / A. Bampakos, P. Deeleman. Birmingham: Packt Publishing, 2023.
4. Freeman, A. Pro React 16 / A. Freeman. Berkeley, California: Apress, 2019
5. Elliott, E. Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries / E. Elliott. Sebastopol: O'Reilly Media, 2014.
6. Blokdyk, G. Comparison of JavaScript Frameworks: Standard Requirements / G. Blokdyk. Canada: 5STARCooks, 2021.

Поступила 01.12.2025 Принята в печать 08.01.2026

References

1. Herman D. (2013) *Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript.* Boston, Addison-Wesley Publ.
2. Flanagan D. (2021) *JavaScript. The Definitive Guide.* Saint Petersburg, Simvol-Plyus Publ. (in Russian).
3. Bampakos A. (2023) *Learning Angular: A No-Nonsense Guide to Building Web Applications with Angular 15.* Birmingham, Packt Publishing.
4. Freeman A. (2019) *Pro React 16.* Berkeley, California, Apress Publ.
5. Elliott E. (2014) *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. Sebastopol, O'Reilly Media Publ.
6. Blokdyk G. (2021) *Comparison of JavaScript Frameworks: Standard Requirements.* Canada, 5STARCooks Publ.

Received: 1 December 2025 Accepted: 8 January 2026

Вклад авторов / Authors' contribution

Авторы внесли равный вклад в написание статьи / The authors contributed equally to the writing of the article.

Сведения об авторах

Струкова А. А., магистрант, ассист. каф. информационных технологий автоматизированных систем, Белорусский государственный университет информатики и радиоэлектроники

Шилин Л. Ю., д-р техн. наук, проф. каф. информационных технологий автоматизированных систем, Белорусский государственный университет информатики и радиоэлектроники

Адрес для корреспонденции

220013, Республика Беларусь,
Минск, ул. Платонова, 39
Белорусский государственный университет
информатики и радиоэлектроники
Тел.: +375 44 747-22-03
E-mail: al-strukova@mail.ru
Струкова Алина Александровна

Information about the authors

Strukova A., Master's Student, Assistant at the Department of Information Technologies of Automated Systems, Belarusian State University of Informatics and Radioelectronics

Shilin L., Dr. Sci. (Tech.), Professor at the Department of Information Technology of Automated Systems, Belarusian State University of Informatics and Radioelectronics

Address for correspondence

220013, Republic of Belarus,
Minsk, Platonova St., 39
Belarusian State University
of Informatics and Radioelectronics
Tel.: +375 44 747-22-03
E-mail: al-strukova@mail.ru
Strukova Alina