Consistent Updates for Scalable Microservices

DEVORA CHAIT-ROTH, New York University, USA
KEDAR S. NAMJOSHI, Nokia Bell Labs, USA
THOMAS WIES, New York University, USA

Online services are commonly implemented with a scalable microservice architecture, where isomorphic workers process client requests, recording persistent state in a backend data store. To maintain service, modifications to service functionality must be made on the fly – i.e., as the service continues to process client requests – but doing so is challenging. The central difficulty is that of avoiding inconsistencies from *mixed-mode* operation, caused by workers of current and new versions interacting via the data store. Some update methods avoid mixed-mode altogether, but only at the cost of substantial inefficiency – by doubling resources (memory and compute), or by halving throughput. The alternative is an uncontrolled "rolling" update, which runs the risk of serious service failures arising from inconsistent mixed-mode behavior.

Ideally, it should appear to every client that a service update takes effect atomically; this ensures that a client is not exposed to inconsistent mixed-mode behavior. In this paper, we introduce a framework that formalizes this intuition and develop foundational theory for reasoning about update consistency. We apply this theory to derive the first algorithms that guarantee consistency for mixed-mode updates. The algorithms rely on semantic properties of service actions, such as commutativity. We show that this is unavoidable, by proving that any semantically oblivious mixed-mode update method must allow inconsistencies.

CCS Concepts: • **Theory of computation → Distributed algorithms**; **Logic and verification**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: On-the-Fly Update, Dynamic Software Update, Rolling Update, Microservices, Consistency

ACM Reference Format:
Devora Chait-Roth, Kedar S. Namjoshi, and Thomas Wies. 2026. Consistent Updates for Scalable Microservices. *Proc. ACM Program. Lang.* 10, POPL, Article 58 (January 2026), 29 pages. https://doi.org/10.1145/3776700

1 Introduction

Online software services, such as services for email, chat, travel reservations, and the like, are expected to be always available. This requirement prevents such services from being shut down to modify their functionality: the familiar shutdown-update-reboot cycle that one uses to update a personal computer is inapplicable. Service functionality must instead be updated *on the fly*, i.e., as the service continues to process client requests. This work studies on-the-fly updates for the scalable microservice architecture that is common for online services. In a scalable microservice, multiple isomorphic workers process client requests, recording persistent data in a backend data store. This architecture makes it easy to scale the service up or down by controlling the number of active workers. On-the-fly updates are accomplished by shifting requests from non-updated to updated workers without stopping service; precisely how this shift occurs determines performance and functionality during an update.

Authors' Contact Information: Devora Chait-Roth, New York University, Manhattan, USA, dc4451@nyu.edu; Kedar S. Namjoshi, Nokia Bell Labs, Murray Hill, USA, kedar.namjoshi@nokia-bell-labs.com; Thomas Wies, New York University, Manhattan, USA, wies@cs.nyu.edu.

ACM 2475-1421/2026/1-ART58
https://doi.org/10.1145/3776700

A “blue/green” update [13] first builds a full system replica containing the new version. Once this replica is running smoothly, traffic is switched to the replica, and the original system is shut down. But replicating a large system is expensive, difficult, and time-consuming. A “big flip” update [6] avoids replication at the cost of throughput. It shuts down half the workers and updates those, then shuts down the other half, bringing back the original set of (now updated) workers, before updating and bringing back the remainder. Hence the service offers only half of its regular throughput during an update, which may lengthen client request queues and increase the latency of servicing requests.

Why deploy such inefficient updates? The motivation lies in avoiding *mixed-mode* operation: if updated and non-updated workers are concurrently active, they can interact indirectly through the data store, leading to unexpected service behavior and possibly to serious failures. Such risk is taken by a “rolling” update [6], which updates a small number of workers at a time while all other workers remain available. Resources are conserved and throughput reduction is minimal, but, as the service operates in an uncontrolled mixed-mode during the update, the risks are great. A spectacular example is that of Knight Capital, a major financial trading company, that went bankrupt within 45 minutes in 2012 due to a mixed-mode error: new-version messages were incorrectly routed to an old-version worker, triggering an infinite cycle of trades [16]. A recent paper in SOSP 2021 [26] studies 120 distributed systems update failures over the span of 10 years. The authors find that the *majority* of errors were due to incompatible version interactions, and that most of the errors studied were catastrophic.

The challenge of achieving a consistent, efficient, on-the-fly update is thus unresolved. The state of the art is unsatisfactory: one must either accept substantial inefficiencies with blue/green or big-flip updates, or run the risk of serious failures arising from mixed-mode inconsistency in a rolling update. Surprisingly, precise formulations of update consistency and formal reasoning about mixed-mode updates are largely lacking from the literature, precluding the design of provably consistent rolling updates. This work aims to fill the gap. We propose a new and (we believe) natural formulation of update consistency, develop methods for reasoning about the consistency of mixed-mode updates, and use those methods to prove the consistency of new mixed-mode update algorithms.

We begin by proposing a formulation for update consistency. The intuition is that external clients of a service should not be aware of how an update is carried out; in particular, whether it involves mixed-mode operation. Our formulation thus views an update computation as being *consistent* if, from the viewpoint of each client, the service *appears* to update *atomically* from its current to the new version at a globally-quiescent state. A quiescent atomic update, by definition, does not exhibit mixed-mode behavior, so this formulation ensures that any internal mixed-mode operation does not result in behavior that is visible to a client.

To prove update consistency, we introduce an induction principle that relies on rewriting an update computation incrementally, preserving observational equivalence, until a computation with an atomic update is reached. The key is defining a notion of “incremental”; we give a progress measure to quantify how close a computation is to an atomic update.

From this formal framework, we derive the first update algorithms to guarantee update consistency in mixed-mode operation, and formally establish their consistency. These algorithms rely on semantic properties of actions, such as commutativity, which allow any update computation to be systematically rewritten to an atomic update.

We expect the semantic properties of actions to be analyzed prior to an update. However, an update algorithm must consult relevant semantic properties at run time. While this incurs overhead, we show that semantic awareness is necessary: we prove an impossibility result, showing that a semantically oblivious update method cannot avoid inconsistent behavior in mixed-mode.

To summarize, the contributions of this work include:

- A client-focused formulation of update consistency,
- An induction principle for proving update consistency,
- The first consistent mixed-mode update algorithms for scalable microservices, achieved through semantic awareness, and
- An impossibility result showing that semantic awareness is required for consistent mixed-mode updates.

1.1 Scope

Our formulation of update consistency is broadly applicable to general distributed services. However, the update algorithms are specialized to a commonly applied scalable microservice architecture, where client requests are serviced by a set of isomorphic worker nodes operating on a shared data store. (We also use the term "database" in a colloquial sense.) The updates may alter worker behavior arbitrarily but not add new features, as that may require updating clients as well, nor modify the database interface. We assume that only one update is in progress at a time.

We make several assumptions about the behaviors of the microservice processes. First, we assume that clients (as well as workers) are mutually independent: I.e., there is no direct communication between clients nor between workers. Second, we assume that clients are sequential: a client must receive a response to an outstanding request before sending a new request. We similarly assume that workers only service one request at a time. Third, we assume clients treat the service as a black box and are unaware of its internal structure and scale. And fourth, we assume that workers are effectively stateless: they may store temporary state to calculate responses to clients, but that state is not persistent from one request to the next. Connected to this is the simplifying technical assumption that every response calculation accesses the database (possibly doing nothing). These assumptions are met by common services such as email systems, social media platforms, and chat services.

In this work we focus on the foundations of update consistency, using a simple system to more clearly demonstrate our formalization and proof method. In reality, large-scale distributed services may have a network of interacting microservices and may apply updates concurrently to these microservices. We expect that the foundation developed in this work for consistent single-microservice update will lead to provably consistent algorithms for combinations of microservices, but we leave that analysis to future work.

1.2 Prior Work

The consistency of distributed system updates has been studied (intermittently) since at least the early 1980s. Seminal research includes the Argus system [4, 5], DYMOS [10], and CONIC [17]. There, update consistency is formulated in terms of type-safe replacement or preservation of a global service invariant. While these criteria provide a measure of consistency, they are incomplete as, for instance, it is not specified how liveness properties are treated (e.g., whether pending requests must be fulfilled). Nor do they apply to updates that cause behavioral change and thus do not preserve all invariants. Type- and specification-based formulations of consistency are also part of several language-specific formulations of dynamic update for single-server applications (cf. [14, 15, 22, 25]). (We discuss related work more fully in §8.)

The blue/green and big-flip update mechanisms are folklore, although similar methods have been proposed for the Imago system [11] and for concurrent object-based systems [1]. As discussed, consistency is obtained at the price of doubling memory and compute, or by halving throughput.

Network routing configuration updates face similar challenges, and consistency for network updates has been the target of much research (cf. [12] for a recent survey). A strong consistency criterion called *per packet consistency* is introduced in [24]; this requires every network packet to be entirely processed by either the current or the new configuration, never by a mix. The paper proposes an update method that requires doubling router memory. A recent "causal" update method [21] obtains the same strong consistency guarantee while operating in place (i.e., without additional memory), but the algorithm may drop packets to guarantee per-packet consistency. While networks are designed to tolerate packet losses, software systems are less robust, and hence software update methods must be lossless. Applying the causal update method to a microservice may also result in a large number of client requests being dropped.

The consistency formulation and update algorithms developed in this paper avoid these disadvantages. Consistency is based on an external view of the service, not on internal type- or invariant-preservation. The consistency definition requires the update to appear atomic to every client; thus, in determining whether an update would meet service requirements, designers only have to reason about atomic updates, a simpler task. The update algorithms work in place, without additional memory. They are a form of rolling update, which limits throughput loss. The price for these desirable properties is that the system actions must be analyzed for commutativity and other semantic properties. We show that this is unavoidable.

2 Overview

Consider a basic messaging service, where the actions available to clients are sending a message, checking their inbox, and reading a message from their inbox. Every client has an individual inbox in the data store that stores received messages. Client requests are sent to and serviced by worker nodes. For instance, when Alice requests to send a message to Bob, the servicing worker adds the content of the message to Bob's inbox. Then when Bob requests to check his inbox, the servicing worker retrieves his inbox entries and sends that data to Bob. This messaging service is structured as a scalable microservice (§3), allowing workers to be added or removed in accordance with traffic.

We assume two important properties of our clients. First, clients are independent from each other: they do not know the exact sequences of requests sent by other clients, nor the worker responses other clients receive. Certainly, if database accesses caused by two clients' requests overlap, the actions of one can be *indirectly* observed by another. For example, when Alice sends a message to Bob, he can observe the message written to his inbox and deduce that Alice had performed a "send-message" action. But, no client *directly* observes the requests and responses of other clients. The second important client property is that clients are ignorant of the precise actions performed by workers to service their requests. In other words, the internal behavior of the system is a black box to clients. Their observations are restricted to sending requests and receiving responses.

We may want to update the "send" action to attach an automated translation to a message based on the country the recipient is based in. Consider a *rolling deployment* of this update, where workers are updated one by one with all other workers remaining available. Now suppose that during the update, Amelie sends a message from France, in French, to George in England, and Amelie's "send" request is serviced by an updated worker. Much to George's delight, he receives a translation of the message in English rather than needing to translate from French himself. George then sends a message back to Amelie in English, expecting that his message will be automatically translated to French. However, George's "send" request is serviced by a non-updated worker (as may happen in a rolling update), and no French translation is shown, offending Amelie with an English message.

This confusing sequence is caused by the uncontrolled mixture of updated and non-updated workers. George first observes behavior consistent with an updated service, then observes a non-updated behavior (e.g., if Alice sends a response that includes his untranslated message), uncovering the system's internal mixed-mode operation. While this particular inconsistency may be viewed as benign, mixed-mode inconsistencies can have serious consequences, as described in §1. Thus, it is necessary to control the processing of requests during a rolling update.

In this paper, we introduce the first formal framework for reasoning about update consistency. We say that an update is *consistent* if the service behavior observed by each client aligns with an *atomic* update to the service, even though in reality the update process allows concurrent operation of updated and non-updated workers. This formulation of update consistency (§4) lets us better understand what went wrong in our example. George observes that an updated action (Amelie's translated message) occurs before a non-updated action (his untranslated message). From George's perspective, it is impossible to reconsider those events as arising from an atomic service update.

From this framework, we develop the first provably consistent algorithms for rolling updates. Our algorithms exploit the semantic properties of system actions to control and direct client requests such that every client observes an atomic update. Consider again our messaging example. Say Alice sends a message to Bob via an updated worker, and then Bob sends a message to Claire via a non-updated worker. From the perspective of the clients, this trace is indistinguishable from a trace that reverses the order of these operations (since Bob does not check his inbox in between). Both actions write to the database, but regardless of which action is ordered first, they result in the same final database state and client observations. In other words, the two actions *commute*. Bob may assume that his message to Claire was processed before Alice sent her message, with the system updating atomically in between. When a series of actions in a computation commute, they can be repeatedly reordered until we produce a trace with an atomic update (§5). Since clients are unaware of the actual trace beyond their own observations, the reordered trace is equally plausible to clients. It appears to them that the system updated atomically, giving us update consistency.

We can also exploit the semantic properties of *backward*- and *forward-compatibility* to obtain update consistency (§7). Backward-compatibility colloquially refers to instances when new-version workers can service requests in a manner consistent with a non-updated system; forward-compatibility is analogous for old-version workers. Suppose in our messaging service that a "format" action is available to clients to draft a message with special text formatting, like creating a bulleted list. Now suppose that we update the system so that any message sent with an asterisk on a new line is automatically formatted into a bulleted list, without requiring the "format" action first, unless the asterisk is preceded by a backslash. If Alice has observed an updated system and she writes a message with an asterisk on a new line, that request can be serviced by an old-version worker if it is *translated* to an explicit "format" request. Then the old-version worker produces results consistent with an updated system: the request is forward-compatible, which can hide the internal mixed-mode operation from clients. Likewise, if Alice is expecting an old-version response, then her message with an asterisk on a new line can be sent to a new-version worker if it is translated to a message with a backslash; backward-compatibility ensures that the new version worker's response will appear to Alice like that of an old-version worker.

To guarantee consistency, our update algorithms control how client requests are routed to updated or non-updated workers. The commutativity update algorithm (Algorithm 2) enforces that all non-updated worker actions left-commute with all updated worker actions during the update, so that clients may (conceptually) reorder the actual computation to one where all non-updated actions occur before all updated actions. The backward-compatibility update algorithm (Algorithm 3) only directs backward-compatible translations to updated workers, so that clients only observe the update with the final updated worker; the forward-compatibility algorithm (Algorithm 4) flips those conditions. Algorithm 5 combines the commutativity and compatibility update strategies.

We model system processes as communicating labeled transition systems (§3), and update algorithms as yet another transition system which restricts the processes' nondeterministic choice of who to communicate with, when to communicate, and when to update (§4). The core of this paper presents the algorithms and their consistency proofs (§5, §6, §7). While the algorithms are simple to state and to implement, their consistency proofs are surprisingly intricate and it is necessary to perform a careful analysis of several subtle cases. Our consistency proofs rely on a partial order view of computations (§6). In this view, an atomic update becomes a consistent cut of the partially-ordered computation, dividing cleanly between updated and non-updated events. The correctness proofs show that events in any update computation controlled by the algorithm can be systematically reordered to converge on an atomic update. We also prove that any consistent rolling update method must consider the semantics of client requests; any method that is oblivious to semantics will exhibit a mixed-mode inconsistency (§4.2).

3 Formal System Model

In a ***scalable microservice***, such as the messaging service discussed in §2, client requests are serviced by a set of interchangeable workers who operate on a shared database. Clients operate independently and are oblivious to the internal workings of the service. The number of workers can change during operation as traffic demands, making such microservices scalable. In practice, a service implementation includes a load balancer, which ensures that the load from client requests is distributed across workers in a manner that meets certain criteria (e.g., evenly distributed load). As this concerns performance rather than correctness, we ignore the presence of the load balancer and instead imagine client requests being sent to workers directly.

3.1 Scalable Microservices

We start by defining some basic notions. A ***labeled transition system*** (LTS) is a tuple $T = (S, A, \Delta, s_{\mathsf{init}})$ where S is a set of states, A is a set of action labels, $\Delta \subseteq S \times A \times S$ is a set of transitions, and $s_{\mathsf{init}} \in S$ is the initial state. We denote a transition $(s, a, s') \in \Delta$ by $s \xrightarrow{a} s'$.

We model each process in a microservice (clients, workers, and database) as a labeled transition system, where each process can send messages to, and receive messages from, other processes via one-way FIFO channels. Let $\mathcal{P}$ be a finite set of processes and M a set of message values. For $\mathsf{p}, \mathsf{q} \in \mathcal{P}$ and $m \in M$, we write $\mathsf{p.q}!m$ for the action of p sending m to q and $\mathsf{p.q}?m$ for p receiving m from q. The set of p's ***observable actions*** is $O_{\mathsf{p}} = \{\, \mathsf{p.q}!m, \mathsf{p.q}?m \mid \mathsf{q} \in \mathcal{P} \setminus \{\mathsf{p}\}, m \in M \,\}$, along with a disjoint set I_{p} of internal actions. The set of all actions of p is thus $A_{\mathsf{p}} = O_{\mathsf{p}} \cup I_{\mathsf{p}}$.

A ***communicating transition system*** $\mathcal{T}$ over $\mathcal{P}$ and M is a tuple $(\{T_{\mathsf{p}}\}_{\mathsf{p} \in \mathcal{P}}, \mathsf{Chan})$ consisting of a labeled transition system $T_{\mathsf{p}} = (S_{\mathsf{p}}, A_{\mathsf{p}}, \Delta_{\mathsf{p}}, s_{\mathsf{init,p}})$ for every $\mathsf{p} \in \mathcal{P}$ and the set of channels $\mathsf{Chan} \subseteq \{(\mathsf{p}, \mathsf{q}) \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, \mathsf{p} \neq \mathsf{q}\}$. We model a microservice as a communicating transition system over the set of clients, workers, and the database, and over the set of messages (requests, responses, database operations, etc.) sent between them. §3.2 elaborates on the specific architecture we assume.

A ***global state*** σ of $\mathcal{T}$ is a pair $(\mathbf{s}, \xi)$ where $\mathbf{s} \in \Pi_{\mathsf{p} \in \mathcal{P}} S_{\mathsf{p}}$ tracks the local state of each process and $\xi \in \mathsf{Chan} \to M^*$ the channel contents. We denote the projection from a global state σ to the local state $\mathbf{s}(\mathsf{p})$ of an individual process p as $\sigma|_{\mathsf{p}}$. In the initial state $\sigma_{\mathsf{init}} = (\mathbf{s}_{\mathsf{init}}, \xi_{\mathsf{init}})$ of $\mathcal{T}$, each process p is in its initial local state, $\mathbf{s}_{\mathsf{init}}(\mathsf{p}) = s_{\mathsf{init,p}}$, and ξ_{init} maps each channel to the empty sequence ε.

Each transition $\sigma \xrightarrow{a} \sigma'$ of $\mathcal{T}$ has an active process p that performs the action $a \in A_{\mathsf{p}}$, causing the global state to be updated according to the following rules:

- $(\mathbf{s}, \xi) \xrightarrow{\mathsf{p.q}!m} (\mathbf{s}[\mathsf{p} \mapsto s'], \xi[(\mathsf{p}, \mathsf{q}) \mapsto \xi(\mathsf{p}, \mathsf{q}) \cdot m])$ if $(\mathbf{s}(\mathsf{p}), \mathsf{p.q}!m, s') \in \Delta_{\mathsf{p}}$ and $(\mathsf{p}, \mathsf{q}) \in \mathsf{Chan}$.
- $(\mathbf{s}, \xi) \xrightarrow{\mathsf{p.q}?m} (\mathbf{s}[\mathsf{p} \mapsto s'], \xi[(\mathsf{q}, \mathsf{p}) \mapsto w])$ if $(\mathbf{s}(\mathsf{p}), \mathsf{p.q}?m, s') \in \Delta_{\mathsf{p}}$ and $\xi(\mathsf{q}, \mathsf{p}) = m \cdot w$.
- $(\mathbf{s}, \xi) \xrightarrow{\tau} (\mathbf{s}[\mathsf{p} \mapsto s'], \xi)$ if $(\mathbf{s}(\mathsf{p}), \tau, s') \in \Delta_{\mathsf{p}}$ and $\tau \in I_{\mathsf{p}}$.

An ***execution*** of an LTS T (and similarly, of a communicating transition system $\mathcal{T}$) is an infinite alternating sequence of states and actions $\rho = s_0; a_0; s_1; a_1; \ldots$ where for each k, $s_k \xrightarrow{a_k} s_{k+1}$. The ***trace*** of ρ is the sequence of its actions, denoted $tr(\rho) = a_0, a_1, \ldots$. We call ρ a ***computation*** if it starts in the initial state, $s_0 = s_{\mathsf{init}}$. We define a ***computation fragment*** as a finite segment of a computation. A computation fragment is called initial if it starts in s_{init}. We write $\mathcal{L}(T)$ for the set of all traces of T's computations.

For $B \subseteq A$ and a trace α, we denote by $\alpha|_B$ the homomorphic projection of α onto B (i.e., the subsequence of α containing actions in B). This allows us to project out from a computation the actions of a given set. For example, if A is the set $\{a, b, c\}$ and B is the set $\{b, c\}$, the projection of a trace $\alpha = acabacc$ onto B, $\alpha|_B$, is $cbcc$. We use projections throughout the paper to obtain the client observations of a given computation. We lift projection to sets of traces in the expected way.

3.2 Architectural Assumptions

We make certain architectural assumptions about the communicating transition system $\mathcal{T}$ defining a microservice. First, we assume $\mathcal{P} = C \uplus \mathcal{W} \uplus \{\mathsf{d}\}$ where C is the set of clients, $\mathcal{W}$ the set of workers, and d the database process. The communication topology of a microservice is restricted as follows: there exists a channel from every client to every worker, from every worker to the database, from the database to every worker, and from every worker to every client, i.e. $\mathsf{Chan} = C \times \mathcal{W} \cup \mathcal{W} \times \{\mathsf{d}\} \cup \{\mathsf{d}\} \times \mathcal{W} \cup \mathcal{W} \times C$. This sets up an architecture where clients communicate with workers, workers operate on the database, and then workers send responses back to clients. There are no channels between distinct clients nor between distinct workers: clients are ***independent*** from other clients, and workers are ***independent*** from other workers.

Processes can keep local state. Workers' state is divided into instruction state, which determines how a worker computes responses to requests, and general local state. However, we make the assumption that workers' general local state does not persist between service of different requests; such workers are "stateless" in common parlance.

In this work, we are interested in services where clients send requests ***sequentially***. For example, in a messaging service, a client cannot concurrently send and read a message; it can only perform those actions in sequence. For simplicity, we assume that each worker operates on the database when servicing a request. (This can be a skip.)

We can therefore partition the behavior of a system into a set of ***relays***, where each relay consists of a client request sent from the client to the worker, internal worker operations, database access by the worker, and a client response sent from the worker to the client. For instance, a relay in our messaging service might be a client requesting to read a message, her servicing worker finding which portion of the database to read, reading and (temporarily) storing the relevant portion of the database, and sending that data back to the client.

Now that we have described the basic structure of a microservice, we give more detail about the transition systems of the clients, workers, and database. We use CCS-like notation for describing LTS's T_{c}, T_{w}, and T_{d} that provide specifications for the actual client, worker, and database transition systems. These specifications are to be understood in terms of observational refinement. That is, for the actual transition system T_{c} of a client $\mathsf{c} \in C$, we require $\mathcal{L}(T_{\mathsf{c}})|_{O_{\mathsf{c}}} \subseteq \mathcal{L}(T_{\mathsf{c}}(\mathsf{c}))$ and similarly for workers and the database LTS. More precisely, the transition system of a process can deviate from its specification by performing auxiliary internal actions and by constraining the non-deterministic choices of message values in the observable send actions of the specification.

Clients. Clients send sequential requests to workers and receive responses from them:

$$T_{\mathsf{c}} = \mu \mathsf{C}.\ \sum_{\mathsf{w} \in \mathcal{W}, req \in M} \mathsf{w}!req.\ \sum_{rsp \in M} \mathsf{w}?rsp.\, \mathsf{C}\quad .$$

To avoid notational clutter, we leave the fixed active process c implicit in all actions specified by T_{c}, writing e.g. $\mathsf{w}!req$ for $\mathsf{c}.\mathsf{w}!req$.

Workers. Workers receive requests from clients, operate on the database, calculate responses, and send responses back to clients:

$$T_{\mathsf{w}} = \mu \mathsf{W}. \sum_{\mathsf{c} \in C, req \in M} \mathsf{c}?req. \sum_{op \in M} \mathsf{d}!op. \sum_{res \in M} \mathsf{d}?res. \sum_{rsp \in M} \mathsf{c}!rsp. \mathsf{W} \enspace .$$

We require that all worker LTS are isomorphic up to renaming of internal actions and the worker's identities in observable actions. This assumption is necessary to allow arbitrary replacement of workers during service.

Database. The database receives operation requests from workers and performs each operation atomically, changing the state of the database. (This is an abstraction of actual database operation, which may allow multiple concurrent operations, while ensuring atomicity and isolation.)

$$T_{\mathsf{d}} = \mu \mathsf{DB}. \sum_{\mathsf{w} \in \mathcal{W}, op \in M} \mathsf{w}?op. \sum_{res \in M} \mathsf{w}!res. \mathsf{DB} \enspace .$$

For the remainder of the paper, we assume that the scalable microservices under consideration follow the above architectural assumptions. We refer to these simply as *systems.* We fix a system $\mathcal{T}$ for all of the definitions we make throughout the remainder of this section. Two properties of the architecture follow easily from the definitions: (1) every occurrence of an action of $\mathcal{T}$ is part of exactly one relay, and (2) every relay includes exactly one client, one worker, and the database.

3.3 Client Projection and Equivalence

Our definition of update consistency is focused on the external observations of clients. The p-***projection*** of a computation fragment ρ for a process p is $tr(\rho)|_{A_{\mathsf{p}}}$, the homomorphic projection of $tr(\rho)$ onto p's actions A_{p}. For instance, the c-projection of a computation fragment in which a client c requests to read a message will only contain c's lookup request for the message that it sends to a worker and its reception of the worker's response to that request. Similarly, the M-projection of a sequence of actions α is the homomorphic projection of α onto the messages contained in send and receive actions. The (p, M)-projection is the sequential composition of these two projections; the composition isolates the sequence of messages sent and received by process p while hiding the identities of the particular processes p is communicating with.

Since clients have an obscured view of a service, many computations will be indistinguishable to a client as long as the client receives the same response messages to their requests. We say that two computation fragments ρ and ρ' are c-***equivalent*** for client c if they have the same global start state, their (c, M)-projections are equal, and if both ρ and ρ' are finite, they have the same end state. This definition formalizes the idea that clients cannot distinguish the identity of workers.

For example, consider a computation where Alice sends a message to Bob; he checks his inbox and reads that message; and then Charles sends a message to Alice, which she receives. This computation is Alice-equivalent to one where Alice sends the message to Bob, then receives the message from Charles.

4 Consistent Updates

We formalize the notion of an update algorithm and establish that there cannot be a consistent rolling update algorithm that is oblivious to the semantic properties of client requests. Throughout this paper, the assumption is that the current and new versions of the service are, by themselves, free of error. The focus is thus solely on errors that may arise from mixed-version interactions *during* the update process.

4.1 System Updates

We define a ***system update*** as replacing one or more system processes with another process that communicates on identical channels as the original. In this work, we only consider updates to worker processes. A worker update transforms a worker's instruction state and general local state, but does not change channel state nor channel structure, or alter the types of messages sent or received by workers. These simplifications exclude updates that add new features or remove old features, as those require changes to the set of messages.

We treat worker updates as occurring instantaneously through dedicated internal worker actions, $u \in U_{\mathtt{w}} \subseteq I_{\mathtt{w}}$. An update action simply replaces the worker's instruction state. An update computation updates every worker exactly once and must eventually update all workers. For any computation, the shortest prefix containing all update actions is a computation fragment that we refer to as an ***update fragment***, or just "update" for short.

A ***quiescent update*** is an update where workers are updated only in a *wait* state. Formally, a worker state $s \in S_{\mathtt{w}}$ is a wait state of $\mathtt{w}$ if $\mathtt{w}$ accepts client requests in s, i.e., there exists a transition $s \xrightarrow{\mathtt{w.c?}m} s'$ for some $\mathtt{c}$, m, and s'. A quiescent update guarantees that every client request is serviced by either a non-updated worker or an updated worker, never by a mix, and that a worker is never updated while servicing a request. We assume that all updates of the system are quiescent.

An ***atomic update*** is an update where all update actions are in a single contiguous fragment that contains no non-update actions. We make the assumption that service requirements are met by an atomic (quiescent) update. This is a reasonable framing, as such an update corresponds to a shutdown-update-restart computation. These updates are thus at the core of our consistency formulation below.

Definition 4.1. An update in a computation ρ is ***consistent*** if, for every client $\mathtt{c} \in C$, there exists a computation with an atomic update that is $\mathtt{c}$-equivalent to ρ.

An atomic update is trivially consistent by definition. In a consistent update computation, the view of any client is identical to its view on a (possibly different and possibly client-specific) atomic update computation. From the discussion above, the behavior of a service as viewed by a client is correct (for a service-dependent notion of correctness) in an atomic update. Hence, the service behavior is also correct in a consistent update computation, as no client can distinguish the actual update computation (which need not be atomic) from one where the update is atomic.

Note that in a consistent update, distinct clients may perceive the system as updating at distinct points: it is a $\forall\exists$ property rather than an $\exists\forall$ property. The stronger requirement of a single point of update perceived by all clients is unnecessarily restrictive, since clients are independent. All that matters is that to each client, the update appears to have taken place atomically at some moment, but those moments need not agree for different clients.

Consider again the language translation example from §2. In that computation, George reads a message processed by an updated worker, and is subsequently serviced by a non-updated worker. This sequence exposes a non-atomic update: first a worker updated (as revealed by the updated send-message action), and following that a non-updated worker performed an action before eventually updating. There does not exist a George-equivalent computation with an atomic update, because one cannot reorder these actions visible to George.

4.2 Update Algorithms and Impossibility Result

§5 presents several consistent rolling update algorithms. These algorithms rely on an analysis of the semantic properties of service actions. We show here that this semantic awareness is necessary by presenting an impossibility theorem that establishes that no update algorithm allowing mixed-mode computations can be universally consistent. For this proof, it is necessary to formalize what we mean by "update algorithm."

4.2.1 Update Algorithms. A distributed mechanism for deploying an update in practice (inspired by [21]) is to use auxiliary update *shell* processes around each worker and client that control message flow and track update status, along with an update manager component to direct the update system-wide. When the update shell receives an instruction from the update manager to update its worker, the worker is shut down and replaced by an updated worker.

Rather than representing the update shells and manager of an update algorithm as explicit processes in the formal model $\mathcal{T}$ of a microservice, we treat them abstractly as another labeled transition system that controls some of the non-deterministic choices made by clients, workers, and the database. In particular, the transition system controls which workers a client may send a request to while an update is in progress and the order in which the database processes operations sent by workers, abstracting the role of an update manager. Moreover, it controls which internal update action a worker performs and when, abstracting the role of update shells. We also assume a temporary extension of the message set for update-specific communications. We assume that these messages are included in the set of all messages M.

Formally, an ***update algorithm*** for microservice $\mathcal{T}$ is a labeled transition system $\mathcal{U}$ over a set of ***controllable actions*** $A_{\mathcal{U}} \subseteq A_{\mathcal{T}}$. The controllable actions consist of all observable actions (i.e., sends and receives) and all internal worker update actions of $\mathcal{T}$, $A_{\mathcal{U}} = O_{\mathcal{T}} \cup \bigcup_{\mathtt{w} \in \mathcal{W}} U_{\mathtt{w}}$. We say that a computation ρ of $\mathcal{T}$ is controlled by $\mathcal{U}$ if its trace is consistent with a computation of $\mathcal{U}$, $tr(\rho)|_{A_{\mathcal{U}}} \in \mathcal{L}(\mathcal{U})$. We say that $\mathcal{U}$ guarantees update consistency for $\mathcal{T}$ if all computations controlled by $\mathcal{U}$ are consistent.

4.2.2 Impossibility Result. The blue/green and big flip deployment algorithms share one desirable property: their ability to guarantee update consistency does not depend on semantic details of the underlying system $\mathcal{T}$ nor the specific message values observed during an execution of the system. In particular, they do not make decisions about whether to direct a specific request to an updated worker based on the request's identity or its effect on $\mathcal{T}$'s global state. In other words, they offer universally applicable solutions to update consistency. We call such algorithms *oblivious*.

On the other hand, blue/green and big flip updates disallow mixed-mode entirely. This begs the following question: do there exist update algorithms that are oblivious, yet allow mixed-mode computations? We show here that such an oblivious update scheme is not possible.

To state our impossibility result precisely, we must define our terms. We are interested in update algorithms that lead to mixed-mode operation. Intuitively, a computation $\rho = \sigma_0, a_0, \ldots$ is in mixed-mode if there is interaction between updated and non-updated system components (indirectly via the database). Formally, we say that worker $\mathtt{w} \in \mathcal{W}$ has been updated before time k in ρ if there exists $i < k$ such that $a_i \in U_{\mathtt{w}}$. Then ρ is in ***mixed-mode*** if there exist $\mathtt{w}_1, \mathtt{w}_2 \in \mathcal{W}$ and $i < j$ such that worker $\mathtt{w}_1$ has been updated before i, worker $\mathtt{w}_2$ has not been updated before j, and actions $a_i = \mathtt{w}_i.\mathtt{d}!op_1$ and $a_j = \mathtt{w}_2.\mathtt{d}!op_2$ for some database operations $op_1, op_2 \in M$.

We say that traces α and β are equal up to message values if β can be obtained from α by only changing the message values in the send and receive actions in α. An update algorithm $\mathcal{U}$ is ***oblivious*** if for any α, β over $A_{\mathcal{U}}$ that are equal up to message values, $\alpha \in \mathcal{L}(\mathcal{U})$ iff $\beta \in \mathcal{L}(\mathcal{U})$.

With these definitions, we can now state our impossibility result. Intuitively, an oblivious update algorithm cannot exploit knowledge of the particular system or update, and so it cannot direct requests between updated and non-updated workers in a way that ensures that every client perceives an atomic update. Concurrent service by differently-versioned workers can reveal the system's internal inconsistency. Thus, an oblivious update algorithm cannot guarantee update consistency when the computation operates in mixed-mode.

Theorem 4.2 (Impossibility result). *For $|M| \geq 2$, there does not exist an update algorithm that:*

(1) admits a mixed-mode controlled computation for some system,
(2) is update consistent for all systems, and
(3) is oblivious.

Proof. By way of contradiction, assume that $\mathcal{U}$ is an oblivious update algorithm that is update consistent for all systems with controllable actions $A_{\mathcal{U}}$. Assume further that ρ is a mixed-mode computation of some system $\mathcal{T}$ with controllable actions $A_{\mathcal{U}}$ such that ρ is controlled by $\mathcal{U}$. We construct a system $\mathcal{T}'$ with a computation ρ' that is not update consistent but equal to ρ up to message values. Since $\mathcal{U}$ is oblivious, ρ' must also be controlled by $\mathcal{U}$, but this contradicts the assumption that $\mathcal{U}$ is update consistent for all systems.

For clarity of explanation, in this construction we do not reason explicitly about internal actions occurring in ρ that are not worker update actions. These actions are simply treated as implicit skip actions of every state of the constructed local LTS comprising $\mathcal{T}'$.

As $|M| \geq 2$, we let 0 and 1 represent two distinct elements of M. We then define $\mathcal{T}'$ as follows. Each client c consists of a loop that sends a 0 request to some worker and waits for a 0 or 1 response:

$$T_{\mathsf{c}} = \mu \mathsf{C}.\ \sum_{\mathsf{w} \in \mathcal{W}} \mathsf{w}!0.\ \sum_{m \in \{0,1\}} \mathsf{w}?m.\ \mathsf{C}\ \ .$$

Every worker w consists of a loop that either updates the worker or processes a 0 request from some client. When processing a 0 request, if w has not yet updated, it sends operation 0 to the database and otherwise it sends 1. In either case, it waits for a 0 or 1 response from the database, which it echos to the client:

$$\begin{aligned} T_{\mathsf{w}} = \ & \mu \mathsf{W}.\ \textstyle\sum_{u \in U_{\mathsf{w}}} u.\ (\mu \mathsf{W}'.\ \textstyle\sum_{\mathsf{c} \in C} \mathsf{c}?0.\ \mathsf{d}!1.\ \textstyle\sum_{m \in \{0,1\}} \mathsf{d}?m.\ \mathsf{c}!m.\ \mathsf{W}') \\ & + \textstyle\sum_{\mathsf{c} \in C} \mathsf{c}?0.\ \mathsf{d}!0.\ \textstyle\sum_{m \in \{0,1\}} \mathsf{d}?m.\ \mathsf{c}!m.\ \mathsf{W}\ \ . \end{aligned}$$

The database accepts 0 and 1 operations from any worker. If it receives 0 after having previously received 1 from another worker, then it responds with 1, otherwise with 0 (the database can be made passive by shifting activity to the worker):

$$\begin{aligned} T_{\mathsf{d}} = \mu \mathsf{DB}.\ \textstyle\sum_{\mathsf{w} \in \mathcal{W}}\ & \mathsf{w}?0.\ \mathsf{w}!0.\ \mathsf{DB} \\ & + \mathsf{w}?1.\ \mathsf{w}!0.\ (\mu \mathsf{DB}'.\ \mathsf{w}?0.\ \mathsf{w}!1.\ \mathsf{DB}' + \mathsf{w}?1.\ \mathsf{w}!0.\ \mathsf{DB}')\ \ . \end{aligned}$$

Observe that all computations of $\mathcal{T}'$ with an atomic update only contain client responses of the form c.w?0. Hence, the same is true for any computation of $\mathcal{T}'$ with a consistent update. On the other hand, any mixed-mode computation contains at least one client response c.w?1. Therefore, $\mathcal{T}'$ does not have any mixed-mode computation that is update consistent.

Recall that ρ is a mixed-mode computation controlled by $\mathcal{U}$. We now map $tr(\rho) = a_0, a_1, \ldots$ inductively onto a trace $\alpha = b_0, b_1, \ldots$ of $\mathcal{T}'$, a system which by definition cannot admit an update consistent mixed-mode computation. Intuitively, the mapping replaces the message value occurring in each a_k with 0, 1 to obtain b_k in a way that produces a valid execution of $\mathcal{T}'$, leaving all internal actions unchanged. Validity of b_k is guaranteed using auxiliary functions defined over the already constructed prefix $b_0, \ldots, b_{k-1}$ to determine what state $\mathcal{T}'$ will have to be in when executing b_k. For example, when a_k is a receive action $a_k = \mathsf{c.w}?m'$ for some m', the mapping has to inspect the constructed prefix trace to see what value $m \in \{0, 1\}$ was most recently sent from w to c and substitute this for m' in a_k to obtain b_k. Similarly, when a_k is a database response to a worker $a_k = \mathsf{d.w}!m'$, it has to inspect the prefix to determine what w-operation d is responding to and whether d has already seen an operation request from an updated worker.

Formally, the mapping is defined as follows:

$$b_k = \begin{cases} \mathsf{c.w!0} & \text{if } a_k = \mathsf{c.w!}m \\ \mathsf{c.w?}m & \text{if } a_k = \mathsf{c.w?}m' \wedge \mathit{lastsnd}(k, \mathsf{w}, \mathsf{c}) = m \\ \mathsf{w.d!}m & \text{if } a_k = \mathsf{w.d!}m' \wedge \mathit{updated}(k, \mathsf{w}) = m \\ \mathsf{w.d?}m & \text{if } a_k = \mathsf{w.d?}m' \wedge \mathit{lastsnd}(k, \mathsf{d}, \mathsf{w}) = m \\ \mathsf{d.w!1} & \text{if } a_k = \mathsf{d.w!}m' \wedge \mathit{lastsnd}(k, \mathsf{w}, \mathsf{d}) = 0 \wedge \exists \mathsf{w}' \in \mathcal{W}.\ \mathit{lastrcv}(k, \mathsf{d}, \mathsf{w}') = 1 \\ \mathsf{d.w!0} & \text{if } a_k = \mathsf{d.w!}m' \wedge (\mathit{lastsnd}(k, \mathsf{w}, \mathsf{d}) = 1 \vee \forall \mathsf{w}' \in \mathcal{W}.\ \mathit{lastrcv}(k, \mathsf{d}, \mathsf{w}') = 0) \\ \mathsf{d.w?}m & \text{if } a_k = \mathsf{d.w?}m' \wedge \mathit{lastsnd}(k, \mathsf{w}, \mathsf{d}) = m \\ a_k & \text{otherwise} \end{cases}$$

where $\mathit{lastsnd}(k, \mathsf{p}, \mathsf{q})$ is m if the last send action from p to q in $b_0, \ldots, b_{k-1}$ is $\mathsf{p.q!}m$ and 0 if no such send actions exist. The function $\mathit{lastrcv}(k, \mathsf{p}, \mathsf{q})$ is defined correspondingly. Similarly, $\mathit{updated}(k, \mathsf{w})$ is 1 if some $u \in U_{\mathsf{w}}$ occurs in $b_0, \ldots, b_{k-1}$, and 0 otherwise.

By induction, one may construct a computation ρ' of $\mathcal{T}'$ with $tr(\rho') = \alpha$. By construction, ρ' is equal to ρ up to message values. Since ρ is a mixed-mode computation, it follows that ρ' is in mixed-mode, and since it is a computation of $\mathcal{T}'$ it cannot be update consistent. However, because $\mathcal{U}$ is oblivious, ρ' is controlled by $\mathcal{U}$ and, hence, $\mathcal{U}$ is not update consistent for $\mathcal{T}'$. Contradiction. □

5 Commutativity-Based Update Algorithms

In this and the following sections, we give algorithms that guarantee update consistency for mixed-mode updates by exploiting semantic properties of system actions. At a system level, the actions of one client are not directly observed by another client; moreover, from a client's perspective, the service is a black box. This, together with the client-focused view of consistency, opens up possibilities for rearranging or even rewriting service internal actions in a manner that is oblivious to a client, but ensures that she sees results that appear to arise from an atomic update.

5.1 Orderedness

It is essential that any rewrite preserve a given client's observations. Thus, the requests made by a client cannot be serviced in an observable manner by an alternation of updated and non-updated workers, since there will then not exist a client-equivalent computation for that client with an atomic update. We call this property an ***ordered update***: once a client request is serviced by an updated worker, subsequent requests by that client are serviced only by updated workers. One can think of an ordered update as fixing the atomic update point for each client at the moment they receive their first updated response. Since a client observes the update at that moment, orderedness ensures that they will continue to directly observe service by updated workers. More work is required, however, to ensure that their indirect observations via the database also remain consistent with an atomic update from that point onward. Algorithm 1 only enforces orderedness but not update consistency. (In fact, it is oblivious.) Algorithm 2 and Algorithm 5 extend Algorithm 1, exploiting commutativity and controlling service to ensure that observations via the database are consistent with an atomic update for each client.

Orderedness is necessary when the version of a worker is observable. Properties like backward- and forward-compatibility obscure the version of a servicing worker, allowing us to manipulate *when* each client observes an update. This renders orderedness unnecessary and even undesirable, since orderedness *fixes* each client's observed atomic update point with their first updated response. Algorithm 3 and Algorithm 4 exploit backward- and forward-compatibility to shift each client's observed update point later or earlier, respectively, giving update-consistency without orderedness.

Algorithm 1 Orderedness

(1) The update manager sends an “update” instruction to a worker.
(2) When the worker shell receives the update instruction, it waits for the worker to complete servicing its current request (if any), then updates the worker process, and sends an “update complete” message to the update manager.
(3) Upon processing a client request, an updated worker tags the requesting client as “updated” by attaching an update tag to the response message.
(4) On receiving an update-tagged response, a client shell changes its status to “updated” and tags all outgoing requests as arising from an updated client.
(5) The update manager directs all requests from updated clients to updated workers.
(6) The update manager continues to send “update” instructions until all workers are updated.

For our algorithms, we suppose that every system component – clients, workers, and the database – has an associated “update shell” as described in §4.2.1. This is an auxiliary process that controls the flow of messages in and out of the component, determines the point at which the component is updated, and tracks the update status of the component. Although only workers update, client shells can record clients as “updated” after receiving their first updated response to enforce orderedness. Likewise, we assume an “update manager” component that controls the flow of messages from clients to workers and instructs workers to update (via the worker shell). The individual shells and the update manager are activated only for the duration of an update.

Each algorithm requires an analysis of relevant semantic properties of service actions. Since the set of service actions is static, this analysis is meant to be performed in advance of an update and consulted during runtime.

5.2 Commutativity

The first semantic property we exploit to guarantee update consistency is commutativity. Intuitively, if two client requests commute, a client cannot identify the order in which the requests were serviced. In a case where an updated action was performed before a non-updated action, commutativity can allow an observing client to assume that the actions were performed in the opposite order, hiding mixed-mode behavior from the client.

Consider our message translation update once more. Suppose Alice sends a message to Bob via an updated worker, so her message is automatically translated. Before checking his inbox, Bob sends a message to Charles via a non-updated worker, so no translation is attached. Then, Bob checks his inbox and sees the automatically translated message from Alice. Although in the actual computation, Alice’s updated send precedes Bob’s non-updated send, the two actions are commutative: without a check-inbox action by Bob in between, there is no dependency ordering between the two message sends. Thus, Bob may assume that his non-updated message send preceded Alice’s updated message send, giving him the perception of an atomic update.

5.2.1 Formal Definitions of Commutativity. We say that action a ***right-commutes*** with action b if for all initial computation fragments $\alpha; a; \sigma_0; b; \sigma_1$, there exists a state σ_0' such that $\alpha; b; \sigma_0'; a; \sigma_1$ is also a computation fragment. ***Left-commuting*** is defined similarly. Two actions ***commute*** if they both left- and right-commute. We generalize these definitions to commutativity of sequences of actions α and β in the expected way.

5.2.2 Commutativity Algorithm. Algorithm 2 exploits commutativity in a basic manner to ensure update consistency. The algorithm itself is simple and the proof of update consistency for this algorithm seems intuitive, but in fact the formal proof is quite involved. After presenting the

Algorithm 2 Basic Commutativity (Extension of Algorithm 1)

(1) At its start, the update manager notifies the database that the update has begun.
(2) During the update, the update manager tags all incoming client requests with a sequence number. The database enforces that requests are completed in the assigned order.
(3) The update manager operates as follows:
   (a) On receiving a request r from an updated client, the manager sends the request to an updated worker, as defined in Algorithm 1.
   (b) Upon receiving a request r from a non-updated client, the update manager checks if the database transactions of the current-version of r commute to the left of all prior new-version database transactions. If that is the case, the update manager may assign the request r to any worker process, regardless of worker version. Otherwise, the update manager must assign this request to a new version worker.

algorithm and its intuitive proof, we will build up the technical machinery required to prove update consistency, for this algorithm and for our subsequent algorithms.

Note that by the architectural assumptions on systems, every relay contains a computation fragment whose trace δ_r is a sequence of database actions of the shape $\mathtt{d.w}?op.(I_{\mathtt{d}})^*.\mathtt{d.w}!res$. We refer to this trace as the database transaction of the relay (respectively, its associated client request).

Proof Sketch. Intuitively, the commutativity property enforced by Algorithm 2 allows a mixed-mode execution to be sorted after the update is complete, by commuting all non-updated actions to the left (while preserving the ordering between non-updated actions). By commutativity, this (conceptual) sorting guarantees that the end state of the system after sorting is identical to the end state reached by the original update computation. Thus, to every client, the actual system behavior is indistinguishable from that observed during an atomic update.

The proof sketch for the algorithm laid out above is fairly straightforward. However, it is not very precise. A formal proof must justify that every client can perceive a plausible, observationally equivalent computation with an atomic update. For that, we need to define exactly how to "legally" rewrite (informally, to sort) a computation, possibly over multiple rewriting steps, in a manner that preserves client observations. Specifically, in the case of commutativity, this requires us to argue that swapping the order of commuting transactions does not introduce a paradoxical time (formally, dependency) loop, and instead produces a valid computation. In the next section, we develop foundational theory to construct such consistency proofs. We then give a formal proof of the update consistency of Algorithm 2 and present additional algorithms for consistent updates.

Overhead. Algorithm 2 enforces that in an update computation, every non-updated action left-commutes with all preceding updated actions. To achieve this property, we must perform a commutativity analysis of available operations in advance of the update; during the update we must impose an order on all database operations (Step 2), and consult the results of the commutativity analysis (Step 3b). Commutativity analysis can be complex. Several works [3, 9, 23] offer automated techniques for generating commutativity conditions. Running this analysis in advance prevents serious delays during runtime. However, imposing an order on the database can cause serious delays. If the update manager is distributed, assigning an order also requires a consensus protocol.

Ordering database actions is necessary to ensure that all non-updated actions commute with *prior* updated actions, which guarantees update consistency. One option to achieve this property while mitigating delays is for the database to accept any request with a number higher than the previously completed request. Requests with a number lower than the previously completed request are rejected by the database, and sent back to the manager for a new number, re-evaluation of commutativity against all requests with lower numbers, and re-assignment to a worker. (This results in stronger commutativity requirements than necessary, since some of those lower-numbered requests will also be rejected by the database.) Another option is to limit the available operations during the update to ones that are mutually commutative. Then all requests can be sent to any worker, and no database ordering is necessary. This restricts operation during the update, but increases efficiency.

6 Proving Update Consistency

In this section, we develop proof principles for showing the consistency of updates. The definition of update consistency requires that for every client c, the actual computation is c-equivalent to a computation with an atomic update. To show update consistency for a given client, we begin with the original computation and apply a sequence of *client-specific rewrites* that constructively transform the original computation to one with an atomic update. For this argument to be well-founded, each rewrite must reduce a progress measure. We recast consistency in terms of a partially ordered view of a computation that simplifies proofs by focusing only on the required dependencies between actions and define an appropriate progress measure. This leads to an induction principle for consistency proofs.

We fix a communicating transition system $\mathcal{T}$ and an update algorithm $\mathcal{U}$; "computations" refer to computations of $\mathcal{T}$ controlled by $\mathcal{U}$.

6.1 A Partial Order Model

Up until now, we have used an interleaving model for computations, where computations alternate between global states and single actions. However, interleaving is unnecessarily strong as it imposes a total ordering on actions even if they are semantically independent of each other. We now introduce a partial order model that retains only the necessary dependencies between actions to simplify our theorems and proofs. This model is adapted from [18] and extensions such as [8, 20].

We define the set of ordered actions $A^<_{\mathsf{p}}$ of a process p by identifying every occurrence of an action in a computation ρ of T_{p} with the trace of the initial computation fragment of ρ ending with that occurrence. Formally, $A^<_{\mathsf{p}} = \{ \alpha[0,k] \mid k \in \mathbb{N} \wedge \alpha \in \mathcal{L}(T_{\mathsf{p}}) \}$. We use the same symbols to denote ordered actions as we use for system actions whenever clear from context. We also often refer to an ordered action $\alpha[0,k]$ as an action and identify it with the system action $\alpha[k]$. For ordered actions a and b of p, we write $a <_{\mathsf{p}} b$ if a is a prefix of b (informally, a occurs prior to b in the timeline for process p). Let $A^<_{\mathcal{T}} = \bigcup_{\mathsf{p} \in \mathcal{P}} A^<_{\mathsf{p}}$.

A ***po-computation (fragment)*** is a tuple $\varphi = (A, \sigma_0, <)$ where $A \subseteq A^<_{\mathcal{T}}$ is a set of ordered actions, σ_0 is the start state of the po-computation, and $<$ gives a partial order over A such that every total ordering that satisfies the partial order yields a computation fragment starting in σ_0 (when one fills in resulting intermediate states). In other words, a po-computation gives ordering constraints on actions necessary to yield a computation fragment starting in σ_0. We refer to these computation fragments as the linearizations of φ. A well-formed po-computation must have at least one linearization. From this point on, all po-computations discussed are assumed to be well-formed. (The consistency proofs apply to po-computations that are induced by computation fragments, which thus have non-empty sets of linearizations and are therefore well-founded.)

We are interested in po-computations whose partial order is defined by the ***happens-before*** relation [18] $<_{hb}$, where $a <_{hb} b$ if:

- a is a send action and b is its corresponding receive action ("message-passing order"), or
- a and b are both performed by the same process p, and $a <_{\mathsf{p}} b$ ("program order"), or
- $a <_{hb} b$ via the transitive closure of message-passing order and program order.

This relation captures the causal ordering of actions in the system. From this point onward, we assume that all po-computations use the happens-before relation as the partial order, and denote this relation simply as $<$. We lift the happens-before order to sequences of actions, $\alpha < \beta$, to mean that α and β are totally ordered by happens-before and all actions in α happen before actions in β. Note that if the set A in a po-computation is finite, then all linearizations under the happens-before order have the same last state.

Every computation fragment ρ induces a unique po-computation φ via the happens-before relation such that ρ is one of φ's linarizations, but all causally independent actions in ρ are left unordered in φ. The following lemma summarizes a useful property of po-computations. (Proofs omitted from this section can be found in the extended version [7].)

Lemma 6.1. *Suppose a and b are two actions by distinct processes p and q, with $a < b$. Then, there must exist some actions a' and b' such that a' is a send, b' is a receive, and $a \leq_{\mathsf{p}} a' < b' \leq_{\mathsf{q}} b$. In other words, there must be (possibly indirect) message passing between p and q.*

An ***update for a po-computation*** φ is a set of actions containing all the update actions in φ.

An ***atomic update for a po-computation*** is an update where no update actions are related to each other in the partial order. While this definition may seem surprising, it is interchangeable with the atomic update definition for computations.

Lemma 6.2. *A po-computation has an atomic update iff one of its linearizations has one.*

Proof. Only if: Suppose we have no update actions related to each other in the partial order of a po-computation. Then, there exists a total ordering consistent with the partial order such that all update actions occur in one uninterrupted block.

If: Suppose a computation has an atomic update. By definition, all update actions occur in one contiguous block. Consider update actions a and b. We show by contradiction that they must be unrelated in the lifted po-computation. Assume that $a < b$. Update actions are not send/receive actions, nor can we have two updates by one process. By Lemma 6.1, we must then have actions a', b' such that $a < a' < b' < b$ where a' is a send and b' a receive. But that contradicts the assumption that a and b appear in an update-only block in the computation. Hence, all pairs of updates must be unrelated in the lifted po-computation. □

We lift p-equivalence and update consistency to po-computations in the expected way.

Theorem 6.3. *Suppose we have a computation ρ with an update ρ_u and φ_u is the po-computation induced by ρ_u. If for every client c there exists φ'_u that is c-equivalent to φ_u and has an atomic update, then ρ is update-consistent.*

Proof. By definition, ρ is update-consistent if, for every client c, there exists a computation with an atomic update that is c-equivalent to ρ. Let c be a client and φ'_u a po-computation that is c-equivalent to φ_u and has an atomic update. By assumption and Lemma 6.2, there exists a linearization ρ'_u of φ'_u that is an atomic update. Replacing ρ_u with ρ'_u in ρ (possible as both end in the same global state by equivalence) we obtain a computation with an atomic update that is c-equivalent to ρ. □

6.2 Cuts

While partial order models describe a microservice in terms of causality rather than time, we can reference a particular moment in time in a po-computation $\phi = (A, \sigma_0)$ by choosing a point on the (totally ordered) timeline of each process. A ***cut*** [2] is a set of actions $C \subseteq A$ that includes, for each process, all actions by that process up until a certain point. C is ***consistent*** if it is downwards-closed

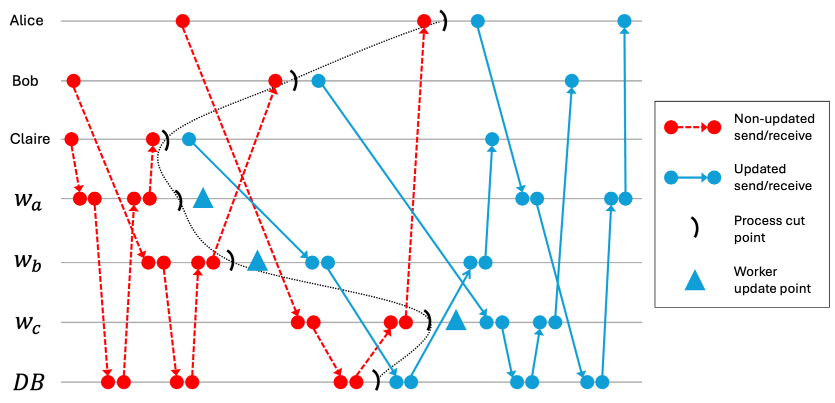


Fig. 1. Space-time diagram showing a global update cut (§6.3), the downwards closure of clients' non-updated receives. Each process's timeline orders actions from left to right. Alice, Bob, and Claire are clients, $w_{a,b,c}$ are workers, and DB is the database. The dotted curve connecting the cut points gives the full cut. The cut is update-consistent: all non-updated actions lie within the cut, while all updated actions lie outside the cut.

according to the happens-before relation. Consistent cuts in φ correspond to reachable states of a corresponding interleaved execution.

Consistent cuts are a useful tool for proving that a po-computation has an atomic update. Although only workers undergo update, we call an action "updated" if it is part of a relay with an updated worker, and "non-updated" otherwise. We define an ***update-consistent cut*** as a consistent cut that includes all the non-updated actions and none of the updated actions of every process. (See Fig. 1. For easier visualization, we assign colors to actions: non-updated actions are colored red and updated actions are colored blue.) An update occurs atomically at a certain point in time if all prior actions are non-updated, and all subsequent actions are updated. An update-consistent cut divides actions in this way, representing an atomic update point.

THEOREM 6.4. *A po-computation has an atomic update iff it admits an update-consistent cut.*

PROOF. Only if direction: let φ be a po-computation that has an atomic update. Choose as the cut the set of all non-updated actions of φ. This cut is consistent by the assumption that φ has an atomic update.

If direction: Suppose the po-computation does not have an atomic update. Then there exists a pair of update actions, a and b, such that $a < b$. By Lemma 6.1, we must then have actions a' and b' where $a < a'$ by program order, $b' < b$ by program order, and $a' < b'$. By assumption, we have an update-consistent cut. Since a' is an updated action, it is not in the update-consistent cut, and since b' is a non-updated action, it is in the update-consistent cut. But since $a' < b'$, the cut is not downwards-closed, contradiction. □

6.3 Progress Measures

Now that we have rebuilt our formal framework in the partial order model, we can use the notion of cuts to give a measure for how "close" a computation is to having an atomic update. An intuitive way for measuring this is by observing the point at which all workers are updated, and noting how many actions prior to that point revealed that the update was not atomic. Every action that revealed the non-atomicity of the update brings the update "farther" from an atomic one.

To capture this expected atomic update point, we define a ***global update cut*** of a po-computation φ, which is the downwards closure of the set of all clients' non-updated receive actions in φ. (See Fig. 1.) This collects all of the actions performed by the system processes up until every client receives their last non-updated response, which will include all non-updated actions.

LEMMA 6.5. *All red actions are inside the global update cut.*

For the global update cut to be perceived by clients as an atomic update point, the cut should be update-consistent, with only non-updated actions before this point and only updated actions after.

Definition 6.6. The ***cut progress measure*** of a po-computation is the number of updated actions inside its global update cut.

THEOREM 6.7. *If the cut progress measure of a po-computation is 0, then it has an atomic update.*

The cut progress measure by itself is insufficient. Two cuts may both have cut progress measure 1, showing that each includes exactly one updated action. But in one cut, the updated action may be closer to the cut point on its process timeline than the other updated action is to its timeline cutpoint. That is, the first cut is closer to being "sorted" than the second. The second progress measure addresses such situations, by measuring sortedness on the timeline of *database* actions. The database is the single point of interaction between distinct workers and clients. Thus, the database exposes the system's mixed-mode operation, and suffices for the sortedness measure.

Definition 6.8. For each non-updated *database action d* in a po-computation φ, let n_d be the number of updated database actions that precede d. The ***sort progress measure*** of a po-computation is the sum of all n_d. (In other words, the sort measure is the number of *inversions*, where non-updated actions should precede updated actions.)

Proving that the sort progress measure suffices for update-consistency requires this lemma:

LEMMA 6.9. *Suppose blue action b is inside of the global update cut. Let rr denote a red client receive action such that b is in the downwards closure of rr. Then b and rr are serviced by distinct workers.*

THEOREM 6.10. *If the sort progress measure of a po-computation is 0, then it has an atomic update.*

Theorem 6.10 relies on our architectural assumptions that clients are sequential, and that every relay contains one database access (modeling no accesses with a "skip" access). These assumptions allow us to use the database as a proxy for the full microservice, since it represents the actions of each client, and covers every relay.

Both the sort and the cut progress measures capture update atomicity at measure 0, and so we can rely on either of them to show update atomicity after repeated decrements. We combine the two measures lexicographically, with the cut measure having higher priority.

Definition 6.11. The ***progress measure*** of a po-computation φ, denoted $pm(\varphi)$ is the lexicographically ordered pair of the cut progress measure and sort progress measure of φ.

6.3.1 Induction Principle. We are now equipped with a progress measure to capture how close a computation is to having an atomic update. Because the ordering on the progress measure is well-founded, this gives us an induction principle for proving update consistency.

Definition 6.12. A ***client rewrite strategy*** $(\mathcal{I}, R)$ consists of a *rewrite invariant* $\mathcal{I}$ and a po-computation transformation R, parametric in a client c such that: (1) $\mathcal{I}$ is a set of po-computations that includes those controlled by $\mathcal{U}$, and (2) for every client c and any po-computation $\varphi \in \mathcal{I}$ where neither component of $pm(\varphi)$ is 0, $R_{\mathsf{c}}(\varphi)$ produces a po-computation $\varphi' \in \mathcal{I}$ that is c-equivalent to φ, ends in the same global state, and reduces the progress measure, i.e., $pm(\varphi') < pm(\varphi)$.

Theorem 6.13. *If there exists a client rewrite strategy, then $\mathcal{U}$ is update-consistent for $\mathcal{T}$.*

Proof. Let $(\mathcal{I}, R)$ be a client rewrite strategy and c be a client. We show by induction on the progress measure that every $\varphi \in \mathcal{I}$ is update-consistent for c.

The base case is where $pm(\varphi)$ is 0 in one of its components. Then φ has an atomic update by either Theorem 6.7 or Theorem 6.10, and is hence update-consistent.

Otherwise, let $\varphi' = R_{\mathsf{c}}(\varphi)$ be the po-computation obtained through the rewrite strategy applied to φ. We then have $pm(\varphi') < pm(\varphi)$ and $\varphi' \in \mathcal{I}$. By the induction hypothesis, φ' is update-consistent for c. That is, for client c, there is a po-computation φ'' that has an atomic update and is c-equivalent to φ'. As φ' is c-equivalent to φ, it follows that φ and φ'' are c-equivalent.

As this applies to every client c and $\mathcal{I}$ contains all po-computations controlled by $\mathcal{U}$, algorithm $\mathcal{U}$ is update-consistent by Theorem 6.3. □

6.4 Commutative Rewrites

We now prove consistency of Algorithm 2. The proof relies on the algorithm's orderedness, which gives the following property.

Lemma 6.14. *Suppose we have an ordered update with some blue action b inside of the global update cut. Let rr denote a red client receive action such that b is in the downwards closure of rr. Then b and rr service distinct clients.*

Theorem 6.15. *Algorithm 2 guarantees a consistent update.*

Proof. We establish this through a rewrite strategy $(\mathcal{I}, R)$ defined as follows. The rewrite invariant $\mathcal{I}$ consists of all po-computations φ such that (1) φ is ordered (i.e., each client timeline consists of non-updated actions followed by updated actions), and (2) every non-updated database transaction in φ is left-commutative with all prior updated database transactions. Consider a computation φ controlled by the algorithm. Step 1 of the algorithm ensures property (1). Moreover, by Step 2 of the algorithm, the update manager is aware of the total ordering on the database, and so Step 3b guarantees property (2). Hence, $\varphi \in \mathcal{I}$.

The po-computation transformation R is then defined as follows. Consider a computation $\varphi \in \mathcal{I}$ and a client c. If neither component of the progress measure is 0, then there must be some updated database transaction in φ which precedes a non-updated database transaction (contributing towards a non-zero sort measure), and there must be some such adjacent pair. Call this updated transaction δ_b and this non-updated transaction δ_r. By property (2) of $\mathcal{I}$, δ_r left-commutes with δ_b. The rewrite $R_{\mathsf{c}}(\varphi)$ exchanges the order of the two transactions.

From the formal model of the database process in §3.2, we can consider the database to act atomically in receiving a request from a worker, acting on it, and sending back a response. We therefore treat δ_r and δ_b as if they were atomic (ordered) actions rd and bd, respectively.

We need to show that this exchange: (I) produces a valid c-equivalent po-computation, (II) strictly decreases the progress measure, and (III) preserves the rewrite invariant $\mathcal{I}$. The claim then follows from Theorem 6.13. We tackle the required conditions in reverse order.

III. Preservation of $\mathcal{I}$. First, left-commutativity is a property of actions and does not depend on a specific computation. Moreover, the exchange only decreases the set of pairs that need to left-commute. Hence, property (2) of $\mathcal{I}$ is preserved. Second, the exchange does not modify the client timelines. Hence, property (1) is preserved.

II. Progress measure decreases. The reordering clearly strictly decreases the sort measure. Thus, we have to show that the cut measure does not increase. To do so, we show that the cut C'

after reordering is a subset of the cut C prior to reordering. Hence, the number of blue (updated) actions in C' is at most the number of updated actions in C.

Consider an action y in C'. By the definition of the cut, action y is in the downward closure of the non-updated actions of clients. (Recall that this update is ordered, so a client timeline consists of non-updated actions followed by updated actions.) Thus, there is a chain of dependencies $\gamma = y < d_1 < \ldots d_k < r$ that links y to some non-updated receive action r on some client c. We consider several cases. (Recall that a database transaction consists of three actions: receiving the operation from a worker, performing the operation on the database, then returning the result to the sending worker.)

(A) γ does not contain events from either rd or bd. In this situation, the chain γ also exists in the original computation; thus, y is in C.

(B) γ contains only events for rd but not bd. Then γ has the form $\alpha; m; rd; \beta$ where m is the previous event on the database timeline or the form $\alpha; rd; \beta$ where the last event of α is on a worker timeline. In the first case, one can construct $\delta = \alpha; m; bd; rd; \beta$; in the second, let $\delta = \gamma$. In each case, the chain δ exists in the original computation, so y is in C.

(C) γ contains events for bd. We can show that γ must contain a database action, say x, that is beyond bd in γ; hence, as $y < rd$ or $y < bd$ in φ, we can form the chain $y < x < r$ in C. See the extended version [7] for details.

I. Produces valid c-equivalent po-computation. The interchange does not affect the actions of the client c, and by commutativity, it leaves the final global state unchanged, so the transformed ordering is c-equivalent to the original.

The difficult part of the proof is to establish that the transformed ordering is a valid computation – i.e., that the transformation does not create a "time loop"; more formally, a cycle of dependencies. This requires a case analysis.

We say "directly causally related" to mean two actions that are either consecutive actions on the same process, or a send action and its corresponding receive action. All other relations in the partial order are from the transitive closure of those relations, hence "indirectly" related. In this proof, we treat a cycle as composed of a sequence of directly causally related actions.

The proof is by contradiction. Suppose that flipping the partial order to obtain $rd < bd$ does induce a cycle in the order. We show that any such cycle would induce a cycle in the original computation, which is impossible. We perform case analysis on the possible events in the cycle.

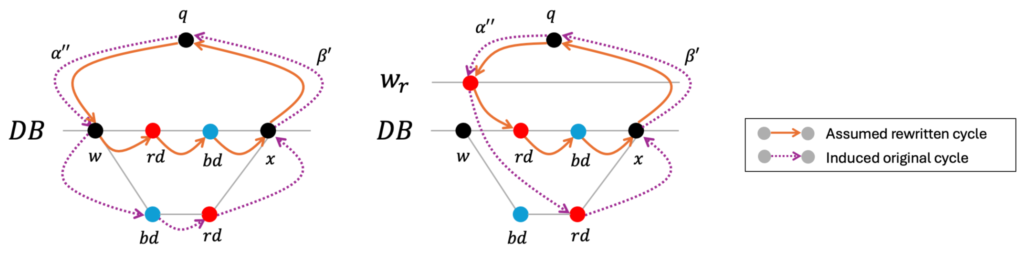
(A) Neither *rd* nor *bd* is in the cycle. The cycle must have been in the order originally, contradicting the assumption that we begin with a proper partial order.

(B) *bd* is in the cycle. For a cycle to exist, we must have a sequence $\gamma = q; \alpha; bd; \beta; q$, where α and β are sequences of actions and q is an action.

Action bd is a database action. By direct causal order , the last action of α must be either the preceding database action, or the send action of the worker (say, wb) who sent the operation request for bd to the database. By assumption, the preceding database action is rd. Likewise, the first action of β must be either the next database action (say, x), or the receive action of wb receiving the results sent by the database in action bd. This gives us a total of four possible cycle shapes for γ (where α' is α without its last action and β' is β without its first action):

(1) $q; \alpha'; rd; bd; x; \beta'; q$
(2) $q; \alpha'; rd; bd; wb\text{-receive-}bd; \beta'; q$
(3) $q; \alpha'; wb\text{-send-}bd; bd; x; \beta'; q$
(4) $q; \alpha'; wb\text{-send-}bd; bd; wb\text{-receive-}bd; \beta'; q$

We proceed by case analysis on the above cycle shapes.



(a) Subcase (1a)   (b) Subcase (1b)

Fig. 2. Illustrations of Case (1) in the proof of Theorem 6.15, Condition I, Part (B). The main database timeline depicts the rewritten computation with $rd < bd$; the original order of $bd < rd$ is depicted below it.

Case (1) $q; \alpha'; rd; bd; x; \beta'; q$. The last action of α' can be either the preceding database action (say, w), or the send action of the worker (say, wr) who sent the operation request for rd to the database. This gives us two subcases (where α'' is α' without the last action).

Subcase (1a) $q; \alpha''; w; rd; bd; x; \beta'; q$. See Fig. 2a. Let us consider the subsequence $w; rd; bd; x$. Action w precedes rd and bd, and so is unaffected by the order swap. By assumption, the state of the database after $rd; bd$ is equal to the state of the database after $bd; rd$ (the original order). Therefore, database action x is identical in either ordering (along with all subsequent actions). The order of rd and bd are swapped in the original, with $w < bd < rd < x$, so the original order must have the cycle $\delta = q; \alpha''; w; bd; rd; x; \beta'; q$.

Subcase (1b) $q; \alpha''; wr\text{-send-}rd; rd; bd; x; \beta'; q$. See Fig. 2b. This subcase is similar to the argument in the previous subcase. The only causal relation that changes in the original ordering is that bd is no longer greater than wr-send-rd, since it is no longer greater than rd. This gives us that in the original order we have the cycle $\delta = q; \alpha''; wr\text{-send-}rd; rd; x; \beta'; q$, where bd is removed from the cycle but the cycle is otherwise unchanged.

Case (2) $q; \alpha'; rd; bd; wb\text{-receive-}bd; \beta'; q$. We again have two subcases.

Subcase (2a) $q; \alpha''; w; rd; bd; wb\text{-receive-}bd; \beta'; q$. By reasoning similar to that in the first case, we have that the original ordering must have the cycle $\delta = q; \alpha''; w; bd; wb\text{-receive-}bd; \beta'; q$.

Subcase (2b) $q; \alpha''; wr\text{-send-}rd; rd; bd; wb\text{-receive-}bd; \beta'; q$. This is the most complex case. For clarity, we shift the start of the cycle to obtain: $bd; wb\text{-receive-}bd; \beta'; q; \alpha''; wr\text{-send-}rd; rd; bd$.

In the original ordering, we have $bd < rd$, but the direct causal relations between all other actions in the cycle remain the same. Thus, we have $bd; wb\text{-receive-}bd; \beta'; q; \alpha''; wr\text{-send-}rd; rd$ in the original computation. But, we do not have $rd < bd$ to create the cycle. It appears that the cycle induced by the swap does not exist in the original computation.

Let us examine the cycle induced by the swap more closely. The actions in the sequence $\beta'; q; \alpha''$ create the relation $wb\text{-receive-}bd < wr\text{-send-}rd$, indirectly relating $bd < rd$. We then obtain a cycle when we swap $bd; rd$ on the database to $rd; bd$: we have $rd < bd$ from the swap and $bd < rd$ from the indirect ordering.

What actions could be present in $\beta'; q; \alpha''$ to create the indirect relation $wb\text{-receive-}bd < wr\text{-send-}rd$? By Lemma 6.9, wb and wr are distinct. So, the causal relation cannot arise from wb-receive-bd and wr-send-rd occurring on the same process; it must arise through send/receive messages between processes. The system architecture dictates that distinct workers can only be indirectly related through interactions with the same client, and through interactions with the database. By assumption, bd and rd are consecutive database actions. So, $\beta'; q; \alpha''$ cannot contain additional database actions to relate $bd < rd$; then we would have database actions in between

bd and rd, which contradicts our assumption that they are consecutive. (These database actions cannot be to the left or the right of $bd; rd$ as that would form a cycle in the original φ.)

Therefore, the indirect relation $bd < rd$ must arise from interactions with the same client. Since this is an ordered update by definition of $\mathcal{I}$, we can apply Lemma 6.14 to conclude that wr and wb are servicing distinct clients. So, the interactions that indirectly relate $bd < rd$ must be between these clients, or via a third distinct client. By assumption, all clients are independent, and only interact with other clients indirectly via the database. Then to relate $bd < rd$ via interactions between distinct clients, we would require database actions in between to relate the distinct clients. (These actions cannot be to either side of $bd; rd$ as that would create a cycle in the original φ.) But this again contradicts our assumption that bd and rd are consecutive actions on the database.

Case (3) $q; \alpha'; wb$-send-$bd; bd; x; \beta'; q$. From an analysis similar to the first case, this would induce a cycle $\delta = q; \alpha'; wb$-send-$bd; bd; rd; x; \beta'; q$ in the original computation.

Case (4) $q; \alpha'; wb$-send-$bd; bd; wb$-receive-$bd; \beta'; q$. This cycle is unaffected by the swap and must exist in the original order.

(C) Only rd **is in the cycle.** Now, suppose rd is part of the cycle induced by the swap. This has the shape $\gamma = q; \alpha; rd; \beta; q$, where α and β are sequences of actions and q is an action. There are two cases to consider:

(1) $q; \alpha'; x; rd; wr$-receive-$rd; \beta'; q$. In this case, we have $\delta = q; \alpha'; x; bd; rd; wr$-receive-$rd; \beta'; q$ is a cycle in the original po-computation.
(2) $q; \alpha'; wr$-send-$rd; rd; wr$-receive-$rd; \beta'; q$. Here, we have that same cycle is also present in the original po-computation.

As any cycle present in the reordered computation induces a cycle in the original computation (which is impossible as the original computation is a partial order), we can conclude that the reordered dependencies cannot result in time loops. □

7 Compatibility-Based Update Algorithms

In this section, we will present further algorithms that guarantee update-consistency by ensuring that actions in a computation can be rewritten – not just reordered – into client-equivalent atomic update computations for each client. These algorithms rely on the semantic properties of backward- and forward-compatibility, both alone and in combination with commutativity.

7.1 Compatibility Algorithms

We present two algorithms for update consistency based on backward- and forward-compatibility, which exploit clients' mutual independence and their ignorance of internal service behavior.

Colloquially, when an update is backward-compatible, old-version requests can be serviced by the updated system in a manner consistent with the original version. For the purpose of semantic analysis for update consistency, we will discuss backward-compatibility at the level of individual requests rather than an entire update. We say that a request is *backward-compatible* if service by an updated worker appears identical to service by a non-updated worker from a client's perspective.

Similarly, forward-compatibility colloquially refers to when new-version requests can be processed by non-updated workers (though this is often limited to coherent error messages). We say that a request is *forward-compatible* if service by a non-updated worker appears identical to service by an updated worker from a client's perspective.

We can achieve compatibility via a *translation layer* by the update manager, if updated actions can be simulated by a sequence of non-updated actions or vice versa. For example, suppose that in our messaging service, a user can include a bulleted list in their message by manually sending a "format" request. We update the service to automatically create a bulleted list format when users

Algorithm 3 Backward-Compatibility

(1) The update manager operates as follows:
On receiving a request r from a non-updated client, the update manager checks if r is backward-compatible under a given translation B. If it is, then the update manager may assign $B(r)$ (treated atomically) to a new worker process or r to an old worker process. Otherwise, r must be assigned to an old version worker process.

type an asterisk on a new line, unless the asterisk is preceded by a backslash. An update manager can enforce forward-compatibility by translating requests with an asterisk on a new line to explicit "format" requests, so that requests serviced by non-updated workers appear to be updated. Likewise, it can enforce backward-compatibility by inserting a backslash to prevent the asterisk from being read as a format request, so that requests serviced by updated workers appear to be non-updated.

Both compatibility definitions give a client-oriented view of compatibility, and we will exploit these semantic properties to guarantee update consistency.

7.1.1 Formal Definitions of Compatibility. Assume that we are given a *backward-translation* partial function $B : M \longrightarrow M^*$ that maps requests to sequences of requests. A request r is ***backward-compatible under translation*** if an updated worker servicing the sequence of requests $B(r)$ (atomically) starting in global state σ results in the same final global state and client projections as a non-updated worker servicing r in global state σ.

Analogously, assume $F : M \longrightarrow M^*$ is a *forward-translation* partial function that maps requests to sequences of requests. A request r is ***forward-compatible under translation*** if a non-updated worker servicing the sequence of requests $F(r)$ (atomically) starting in global state σ results in the same final global state and client projections as an updated worker servicing r in global state σ.

7.1.2 Backward- and Forward-Compatibility Algorithms. Unlike Algorithm 2, the compatibility algorithms do not enforce an ordered update, so that the perceived point of atomic update can be manipulated for update consistency. Algorithm 3 takes advantage of backward-compatibility to ensure update consistency by requiring that only backward-compatible actions are sent to updated workers. The update then appears atomic with the last updated worker. (The algorithm tests whether a request is backward-compatible as the translation function is partial.)

Algorithm 3 can be flipped to ensure update consistency by requiring that only forward-compatible actions are sent to non-updated workers (Algorithm 4), to give the appearance that the system updates atomically with the first updated worker.

Algorithm 4 Forward-Compatibility

(1) The update manager operates as follows:
On receiving a request r from a non-updated client, the update manager checks if r is forward-compatible under a given translation F. If it is, then the update manager may assign $F(r)$ (treated atomically) to an old worker process or r to a new worker process. Otherwise, r must be assigned to a new version worker process.

7.1.3 Compatibility Rewrites. The proof of update consistency for Algorithm 3 follows a similar structure to the proof for Algorithm 2. The proof requires an additional assumption: that the database behavior depends only on the action and not the identity of the requesting worker process. We say that the database is "oblivious to worker identity."

Theorem 7.1. *Algorithm 3 guarantees a consistent update.*

Proof. We will establish this through a rewrite strategy $(\mathcal{I}, R)$ defined as follows. The rewrite invariant $\mathcal{I}$ consists of all po-computations φ such that if there is an updated database transaction δ_b in the relay of client request r which precedes a non-updated database action, then δ_b must be servicing r's backward-compatible translation $B(r)$ rather than r itself. Consider a po-computation φ controlled by the algorithm. Step 1 of the algorithm enforces that every updated database transaction services a backward-compatible translation $B(r)$ of the sent client request r (which is treated atomically). Hence, $\varphi \in \mathcal{I}$.

The po-computation transformation R is then defined as follows. Consider a computation $\varphi \in \mathcal{I}$ and a client $\mathtt{c}$. If neither component of the progress measure is 0, then there must be some updated database transaction δ_b in φ which precedes some non-updated database transaction δ_r (contributing towards a non-zero sort measure). By $\varphi \in \mathcal{I}$, δ_b is servicing a backward-compatible translation $B(r)$ of the sent client request r (which is treated atomically). Let $\mathtt{w}_u$ denote the (updated) worker who services $B(r)$ and let s denote the global state in which $\mathtt{w}_u$ services $B(r)$. The rewrite $R_{\mathtt{c}}(\varphi)$ introduces a fresh non-updated worker $\mathtt{w}_n$ which services request r in global state σ, and $\mathtt{w}_u$ does not service $B(r)$. (In other words, we replace $B(r)$ by $\mathtt{w}_u$ with r by $\mathtt{w}_n$.) This is the only request that $\mathtt{w}_n$ services. Let δ_t denote the non-updated database transaction which services r.

As in the proof of Theorem 6.15, we treat δ_b, δ_t, and δ_r as if they were atomic (ordered) actions bd, t, and rd, respectively.

We need to show that this replacement (I) produces a valid $\mathtt{c}$-equivalent po-computation, (II) strictly decreases the progress measure, and (III) preserves the rewrite invariant $\mathcal{I}$. The claim then follows from Theorem 6.13. We tackle the required conditions in reverse order.

III. Preservation of $\mathcal{I}$. Backward-compatibility is a property of actions and does not depend on a specific computation. Moreover, the rewrite only decreases the set of updated actions servicing backward-compatible translations. Hence, $\mathcal{I}$ is preserved.

II. Progress measure decreases. By Lemma 6.5, every red action is inside the global update cut. Since $bd < rd$ by assumption, bd is inside the global update cut as well, so the cut measure is greater than 0. The substitution replaces (updated) bd with (non-updated) t. This removes one blue action from the global update cut, decrementing the cut measure and hence decrementing the computation's progress measure.

I. Produces valid $\mathtt{c}$-equivalent po-computation. By our assumption that clients do not know the set of active workers $\mathcal{W}$, the rewrite can introduce a new, non-updated worker $\mathtt{w}_n$. The result of worker $\mathtt{w}_n$ servicing the relay produces the same client projections as the original computation for all clients by definition of backward-compatibility, the obliviousness of the database to worker identity, and the fact that clients cannot tell the identities of workers apart. So, this substitution is client-equivalent to the original for all clients.

By definition of backward-compatibility, the final global state resulting from the substituted computation will be equal to the final global state in the original computation. $\square$

The proof for update consistency of Algorithm 4 is analogous.

7.2 Commutativity + Forward-Compatibility Algorithm

We can combine the notions of commutativity and compatibility to derive additional consistent update algorithms. We present Algorithm 5 which exploits commutativity and forward-compatibility; conditions can be flipped for backward-compatibility (see the extended version [7]). Orderedness is enforced in a limited way: when worker version can be obscured by forward-compatibility, orderedness is not enforced. The (translation of) the request can be serviced by a non-updated worker even for an updated client, since the client will perceive that worker as updated. However,

Algorithm 5 Commutativity with Forward-Compatibility (Limited Extension of Algorithm 1)

```
(1) At the start, the update manager notifies the database that the update has begun.
(2) During the update, the update manager tags all incoming client requests with a number. The
    database enforces that requests are completed in the assigned order.
(3) The update manager operates as follows:
    (a) Upon receiving a request $r$, the update manager checks if it is forward-compatible under
        a given translation $F$. If it is forward-compatible, then the update manager may assign $F(r)$
        (treated atomically) to an old worker process, or $r$ to a new worker process.
    (b) If it is not forward-compatible:
        (i) If $r$ is from an updated client, the manager sends the request to an updated worker, as
            defined in Algorithm 1.
        (ii) If $r$ is from a non-updated client, the update manager checks if $r$ commutes to the left
            over all prior new-version requests and old-version forward-compatible translations.
        (iii) If $r$ does left-commute, the update manager may assign $r$ to any worker, regardless of
            version.
        (iv) Otherwise, the update manager must assign $r$ to a new version worker process.
```

an updated client must be serviced strictly by updated workers for requests that do not have compatible translations.

Theorem 7.2. *Algorithm 5 guarantees a consistent update.*

Intuitively, the rewrite invariant $\mathcal{I}$ includes all po-computations such that all non-updated jobs are either forward-compatible or left-commutative, a property enforced by Algorithm 5. This property ensures that non-updated jobs can be perceived as either updated or having occurred earlier, before the first perceived update point. The rewrite strategy R first replaces a forward-compatibility translation $F(r)$ serviced by a non-updated worker with r serviced by an updated worker, and then left-commutes subsequent non-updated database actions past prior updated database actions (including the database actions servicing the newly introduced r).

Note that this rewrite strategy requires a combination of commutativity and compatibility. However, isolating the steps may not lead to a rewrite strategy. Replacing $F(r)$ by an old worker with r by a new worker may not make progress if the database transaction of $F(r)$ is not immediately preceding the final (non-updated) database transaction inside the cut. And the ability to left-commutate an old-version database transaction past prior new-version database transactions is only guaranteed in combination with the compatibility replacement step.

8 Related Work

Type Safety and Invariance Preservation. The seminal research on the correctness of dynamic (on-the-fly) software update includes that on the Argus system [4, 5] and DYMOS [10]. These systems allow the type-safe replacement of a program module that is in a quiescent state, typically under severe constraints to prevent inconsistencies. The work on Argus, for instance, requires that the new module type must be compatible with the old type, in that the new module has the same future behavior as the module it replaces. (E.g., the new module code may include performance improvements but keep the original interface.) Another early system is CONIC [17]. It defines a small set of management commands (e.g., passivate, activate, unlink, link), using which one can program updates to a transaction-based distributed system. The correctness guarantee for the update is in terms of preserving a global invariant.

While invariant or type preservation provides a measure of consistency, such guarantees are necessarily partial in that they do not cover the full behavior of a system; for instance, nothing is said about liveness requirements, such as requiring a proper response to pending requests. Moreover, these internal guarantees do not necessarily imply a consistent view of the update for external clients. A practical difficulty is that few real-world systems have associated formal proofs of invariance (or, more generally, safety); thus, it is a challenge to establish the preservation of type or invariance safety properties for an update.

Language-specific formulations of update consistency include methods such as DSU (dynamic software updates), which are typically for single-server deployments rather than the distributed multi-server setting discussed above and targeted in this paper. Hicks et al. [15], Neamtiu et al. [22], Stoyle et al. [25] develop a calculus and implementation for type-preservation in DSU of C programs, which ensures that any type that is updated is not used concretely after its update point. While helpful, type-consistency is a weak requirement, as it does not prevent clients from observing responses that may only arise from mixed-mode operation during an update. The work in [14] tackles this concern by requiring the correctness definition to be system-specific and by defining a program transformation that merges the old and new versions to model DSU in a manner that suffices for verification. This method and others that rely on verification are often unworkable in practice simply because many services do not have a precise formal specification, the implementation ranges over multiple programming languages, and verification of such large and complex service implementations is often infeasible in practice.

Consistency via Resource Replication. The blue/green and big-flip update mechanisms are folklore methods [6, 13]. Consistency comes at the price of significant inefficiency: a blue/green update requires doubling existing compute and memory resources, while a big-flip update halves throughput during the update. The Imago system [11] follows an approach related to the blue/green method. A full system replica is created, then persistent data is transferred from the original version to the updated replica opportunistically. Once all persistent data has been transferred, traffic is switched to the updated replica. The approach taken in [1] is similar, in that it allows multiple versions of each component to co-exist concurrently during the update. Concurrently active versions of a component must maintain consistency between their internal states; this is handled by defining a state transformation function or an inter-state consistency relation. It might be necessary to drop an incoming request if it cannot be handled consistently by the multiple active versions of an object. Thus, besides requiring additional memory and compute resources, this method may also drop a request at some intermediate point in a chain of related requests. That makes it challenging to preserve global consistency; indeed, the work does not include a formal consistency argument.

Consistent Network Updates. Message drops are less of a concern for updates to network configurations, as network protocols are designed to be resilient to packet drops. A strong consistency criterion for network updates, called *per packet consistency*, is introduced in [24]. This property requires that the route taken by a packet through the network during an update either entirely follows the old configuration or entirely follows the new configuration, never a mix of the two. The paper proposes a two-phase algorithm, where old and new configurations co-exist at each router, with new traffic routed only through the new configurations until all old traffic has been processed. This scheme doubles the memory requirements per router. A recently developed method, called “causal update” [21], performs a network configuration update without additional memory; however, this in-place update process requires packets sent from an old-configuration router to a new-configuration one to be dropped to avoid violating the per-packet consistency requirement.

The causal update algorithm guarantees that the network update appears atomic which, in turn, establishes per-packet consistency. (Per-packet consistency allows one packet from a client to be processed with the new configuration while the next packet from the same client is processed by the old configuration. This is disallowed in the consistency formulation of [21].) Our definition of update consistency also requires a service update to appear atomic but takes a client-based view, allowing different clients to view the service update as having taken place at different points.

Analysis Tools. Short of guaranteeing update consistency, tools such as monitoring systems, testing systems, and static analyzers can be used to minimize damage from update-induced failures. Gandalf [19] is an analysis tool for large-scale systems to catch anomalies or bugs during rollouts of software updates, currently used by Microsoft Azure. DUPTester is a testing framework designed to test upgrades on a small scale so upgrade bugs can be found before deployment; DUPChecker is an accompanying static analysis tool that searches for data syntax incompatibilities [26]. These tools and others can assist in mitigating update failures; we hope that our work can pave the way for provably consistent updates.

Conclusion. The consistency formulation in this paper overcomes key disadvantages of prior methods. Consistency is based on the external behavior of the underlying service as seen by its clients, a view more appropriate for a service. The formulation side-steps the difficulty with the lack of formal service specifications in practice by requiring that each client views an update as being atomic. Thus, system designers have only to check (or enforce) that the desired (possibly informal) specifications are maintained by an atomic update, a simpler and more natural task. The formulation and consistency analysis are independent of the programming languages used to implement a system, and are thus more widely applicable. The proposed update mechanisms work in-place, thus avoiding the disadvantages of the big-flip and blue/green methods. Our algorithms do depend on an analysis of commutativity and other properties of the service actions across versions. While we show that this is unavoidable in general, one does need to perform such analysis. Automated commutativity reasoning for programs is quite challenging (cf. [23]), but it appears to be easier to reason about commutativity at the level of the abstract semantics of service operations, as is done in the (informally analyzed) examples in this paper.

In future work, we plan to take on implementation and optimization concerns, applying our foundations to obtain efficient update algorithms for complex systems. In this vein, an important direction for further research is to develop consistent update methods for services that are designed as a network of microservices: that can perhaps be done through a combination of the single-microservice update methods with the network-level causal update mechanism discussed above. The design of efficient, in-place, provably consistent update methods is, we believe, a largely unexplored topic, one with both substantial practical importance and a rich mathematical structure.

Acknowledgments

This work is supported in parts by the National Science Foundation under the grant agreement 2304758. We thank our anonymous reviewers for their valuable feedback. We thank Aurojit Panda, Sahil Tikale, and Amit Lieber for providing systems and industry perspectives, and Sahil for suggesting the accept-higher-numbers optimization for Algorithm 2. We also acknowledge many helpful discussions with colleagues at Nokia Bell Labs and New York University.

References

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. 2006. Modular Software Upgrades for Distributed Systems. In *ECOOP 2006 – Object-Oriented Programming (Lecture Notes in Computer Science, Vol. 4067)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 452–476. doi:10.1007/11785477_26

[2] Özalp Babaoğlu and Keith Marzullo. 1993. *Consistent global states of distributed systems: fundamental concepts and mechanisms.* ACM Press/Addison-Wesley Publishing Co., USA, 55–96.

[3] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 115–132. doi:10.1007/978-3-319-89960-2_7

[4] Toby Bloom. 1983. *Dynamic Module Replacement in a Distributed Programming System.* Ph. D. Dissertation. MIT, https://dspace.mit.edu/handle/1721.1/15685.

[5] Toby Bloom and Mark Day. 1993. Reconfiguration and module replacement in Argus: theory and practice. *Softw. Eng. J.* 8, 2 (1993), 102–108.

[6] E.A. Brewer. 2001. Lessons from giant-scale services. *IEEE internet computing* 5, 4 (2001), 46–55. doi:10.1109/4236.939450

[7] Devora Chait-Roth, Kedar S. Namjoshi, and Thomas Wies. 2025. Consistent Updates for Scalable Microservices. arXiv:2508.04829 [cs.PL] https://arxiv.org/abs/2508.04829

[8] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. doi:10.1145/214451.214456

[9] Adam Chen, Parisa Fathololumi, Eric Koskinen, and Jared Pincus. 2022. Veracity: declarative multicore programming with commutativity. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 186 (Oct. 2022), 31 pages. doi:10.1145/3563349

[10] Robert P. Cook and Insup Lee. 1983. DYMOS: a dynamic modification system. In *SIGSOFT*. ACM, 201–202. doi:10.1145/1006140.1006188

[11] Tudor Dumitraş and Priya Narasimhan. 2009. Why Do Upgrades Fail and What Can We Do about It?. In *Middleware 2009 (Lecture Notes in Computer Science, Vol. 5896)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 349–372. doi:10.1007/978-3-642-10445-9_18

[12] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2019. Survey of Consistent Software-Defined Network Updates. *IEEE Communications Surveys & Tutorials* (2019). doi:10.1109/COMST.2018.2876749

[13] Martin Fowler. 2010. Blue Green Deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html. Accessed: 2025-09-15.

[14] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. 2012. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Verified Software: Theories, Tools, Experiments (Lecture Notes in Computer Science, Vol. 7152)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293. doi:10.1007/978-3-642-27705-4_22

[15] Michael Hicks, Jonathan T. Moore, and Scott Nettles. 2001. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 13–23. doi:10.1145/378795.378798

[16] Knight Capital 2013. Knight Capital LLC SEC Administrative Proceedings. https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf. Accessed: 2024-10-16.

[17] Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Software Eng.* 16, 11 (1990), 1293–1306. doi:10.1109/32.60317

[18] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563

[19] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. 2020. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 389–402. https://www.usenix.org/conference/nsdi20/presentation/li

[20] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[21] Kedar S. Namjoshi, Sougol Gheissi, and Krishan Sabnani. 2024. Algorithms for In-Place, Consistent Network Update. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 244–257. doi:10.1145/3651890.3672266

[22] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (Ottawa, Canada). 72–83. doi:10.1145/1133255.1133991

[23] Jared Pincus and Eric Koskinen. 2025. An Abstract Domain for Heap Commutativity. In *VMCAI (2) (Lecture Notes in Computer Science, Vol. 15530)*. Springer, 26–49. doi:10.1007/978-3-031-82703-7_2

[24] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference* (Helsinki, Finland) *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 323–334. doi:10.1145/2342356.2342427

[25] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. 2007. Mutatis Mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.* 29, 4 (aug 2007), 22–es. doi:10.1145/1255450.1255455

[26] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 116–131. doi:10.1145/3477132.3483577

Received 2025-07-10; accepted 2025-11-06