State Management Strategies for Stateful Digital Twin Migration in Industrial Scenarios

M. Fogli, C. Giannelli, G. Rossi and C. Stefanelli

2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT), 2025

Cite this as

M. Fogli, C. Giannelli, G. Rossi and C. Stefanelli, "State Management Strategies for Stateful Digital Twin Migration in Industrial Scenarios," *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, Lucca, Italy, 2025, pp. 435-442, doi: 10.1109/DCOSS-IoT65416.2025.00075.

State Management Strategies for Stateful Digital Twin Migration in Industrial Scenarios

Mattia Fogli*, Carlo Giannelli*, Gaia Rossi*, Cesare Stefanelli*,
* *University of Ferrara, Italy*
{mattia.fogli, carlo.giannelli, gaia.rossi, cesare.stefanelli}@unife.it

***Abstract*—While the adoption of digital twins (DTs) in industrial scenarios has been widely investigated, their dynamic migration remains largely unexplored. DTs fundamentally differ from traditional cloud applications. Due to their cyber-physical nature, DTs are often stateful and heavily rely on contextual and historical data. As a result, migration strategies designed for traditional stateless cloud applications may not be suitable for stateful DTs. To the best of our knowledge, no prior work has addressed the challenge of stateful DT migration. To bridge this gap, this work identifies three state management strategies (state replication, state rebuilding, and state sharing), develops six corresponding implementations (DT API, storage relocation, hot start, cold start, storage rebinding, and distributed cache), and experimentally evaluates them in an industrial setting.**

***Index Terms*—digital twins, industrial internet of things, orchestration, stateful migration, state management**

I. Introduction

The advent of the internet of things (IoT) in industrial environments has recently pushed for a pervasive spread of information technology (IT) technologies and standards within the operational technology (OT) domain. In particular, digital twins (DTs), one of the most compelling pillars of this paradigm shift, have found a place in the realm of industrial internet of things (IIoT) applications. Fundamentally, a DT is the virtual counterpart of a physical element [1]. DTs have already been proposed for a variety of IIoT applications, such as job scheduling [2], anomaly detection [3], and zero defect manufacturing [4].

DTs are becoming integral components of cyber-physical systems (CPSs) [5], [6] and are envisioned as foundational building blocks of digital factories [7]. In this regard, containerization has been identified as a key enabling technology [8]. In fact, containerization and container orchestration—already widely adopted in the cloud—are also now gaining momentum in industrial scenarios. However, DTs greatly differ from traditional cloud applications (e.g., web servers) due to their cyber-physical nature, and thus designed to represent a physical counterpart with its own lifecycle. Another important distinction is that DTs are often stateful, as they heavily rely on contextual and historical data. For example, a DT of a production line must interpret a conveyor belt update within the broader context of the current operational state of the whole system. As a result, migration strategies that work well for cloud applications, which are typically designed as stateless applications, may not be suitable for stateful DTs.

It is important to note that incomplete or inconsistent DT state migration can lead to misinterpretation of real-time data, causing delays, bottlenecks, or downtime. More critically, failing to migrate safety-related data—such as interlocks, emergency stops, or hazard monitoring—can compromise protective systems. This increases the risk of equipment-related accidents or injuries. Thus, stateful DT migration is crucial to maintain productivity and ensure worker safety.

Several works have proposed integrating DTs into the container orchestration process. In these works, DTs typically assist container orchestrators in making informed decisions. Therefore, these works focus on how DTs can help make better orchestration decisions rather than on focusing how to orchestrate DTs themselves. A few works have addressed the orchestration of containerized DTs [9], [10], [11], but without exploring the challenge of migrating stateful DTs.

Various works have investigated the problem of state migration of virtual machines (VMs) and containerized applications [12], [13], [14], [15], [16]. Although these works offer valuable insights into how to tackle the problem of state management for stateful DT migration, they do not take into account the cyber-physical nature of DTs. To the best of our knowledge, there is no prior work on stateful DT migration.

The contributions of this work are as follows:

- We identified three state management strategies for stateful DT migration, i.e., *state replication*, *state rebuilding*, and *state sharing*.
- We developed six implementations, two for each identified state management strategy. Specifically, the *DT API* and *storage relocation* are implementations of the state replication strategy; the *hot start* and *cold start* are implementations of the state rebuilding strategy; and *storage rebinding* and *distributed cache* are implementations of the state sharing strategy.
- We experimentally evaluated such implementations in an industrial setting. The evaluation quantitatively determines downtime, total migration time, and resource overhead for each implementation.

The paper is organized as follows. Section II introduces the DT concept, lays out the foundational properties of DTs, and provides the rationale behind this work. Section III discusses the state management strategies that we identified for stateful DT migration. Section IV details our implementations of the identified state management strategies. Section V experimentally evaluates such implementations in an industrial setting.

Section VI gives an overview of related work. Section VII concludes the paper.

II. Background and Motivation

A. Background

While the concept of DT was first introduced more than twenty years ago by Grieves [17], there is still no general consensus around its definition [18], [19], [20]. The interested reader may refer to [21] for a survey on the proposed definitions. As defined by Grieves [1], the DT model consists of three elements: a physical element that exists or will exist in the physical space, a virtual counterpart that exists in the virtual space (the DT), and a connection between them. Henceforth, the term physical twin (PT) refers to the physical counterpart of a DT.

Minerva et al. [22] identified a set of characterizing properties of DTs in the context of IoT. Specifically, a subset of those properties was identified as foundational: representativeness and contextualization, mirroring, and entanglement. In particular, the entanglement property, recently made available as a metric [23], refers to the connection between the DT and its physical counterpart. Ideally, this connection makes the counterparts instantaneously connected. This means that any change in the PT is instantly reflected in the DT, and vice versa. Practically speaking, this cannot actually happen. In fact, moving data over a network or processing data are operations that always take time. As a rule of thumb, the average time elapsed between two changes should be considered the upper limit for updating the counterpart [22]. When this occurs, the two counterparts are said to be entangled.

From a wider point of view, DTs are becoming integral components of CPSs [5], [6] and envisioned as building blocks of digital factories [7]. Siqueira and Davis [8] conducted an in-depth analysis of the state-of-the-art literature on the adoption of microservices for DTs, highlighting how containerization may be an enabling technology. In this regard, various works relied on containerization for DTs deployment in industrial scenarios. For example, Azarmipour et al. [24] developed a dynamic management solution by designing the manufacturing execution system (MES) and programmable logic controllers (PLCs) as compositions of multiple microservices. Bellavista et al. investigated how to apply design patterns commonly used in microservices for adaptive, autonomous, and context-aware DTs [9], developed an entanglement-aware middleware for DTs [10], and explored orchestration of DTs in the cloud-to-edge continuum [11].

B. Motivation

As a mature technology, containerization and container orchestration have already seen widespread adoption in the cloud and they are currently gathering momentum also in DT-based industrial scenarios. However, migration strategies that work just fine for cloud applications may not be suitable for DTs. For example, web servers are typically designed as stateless applications. These applications do not maintain a session state, thus each request is handled independently. A stateless design brings several advantages, such as scalability (an incoming request can be handled by any available server instance), fault tolerance (a server fault does not impact any user session), and simplicity (no need for state management).

In typical industrial scenarios, DTs are inherently stateful due to their dependence on contextual and historical data. For example, consider a DT of a production line. An update originating from a conveyor belt cannot be processed in isolation but must be interpreted within the broader context and current operational state of the entire production system. Each event or update impacts and depends on preceding states, underscoring the importance of maintaining a comprehensive state within the DT.

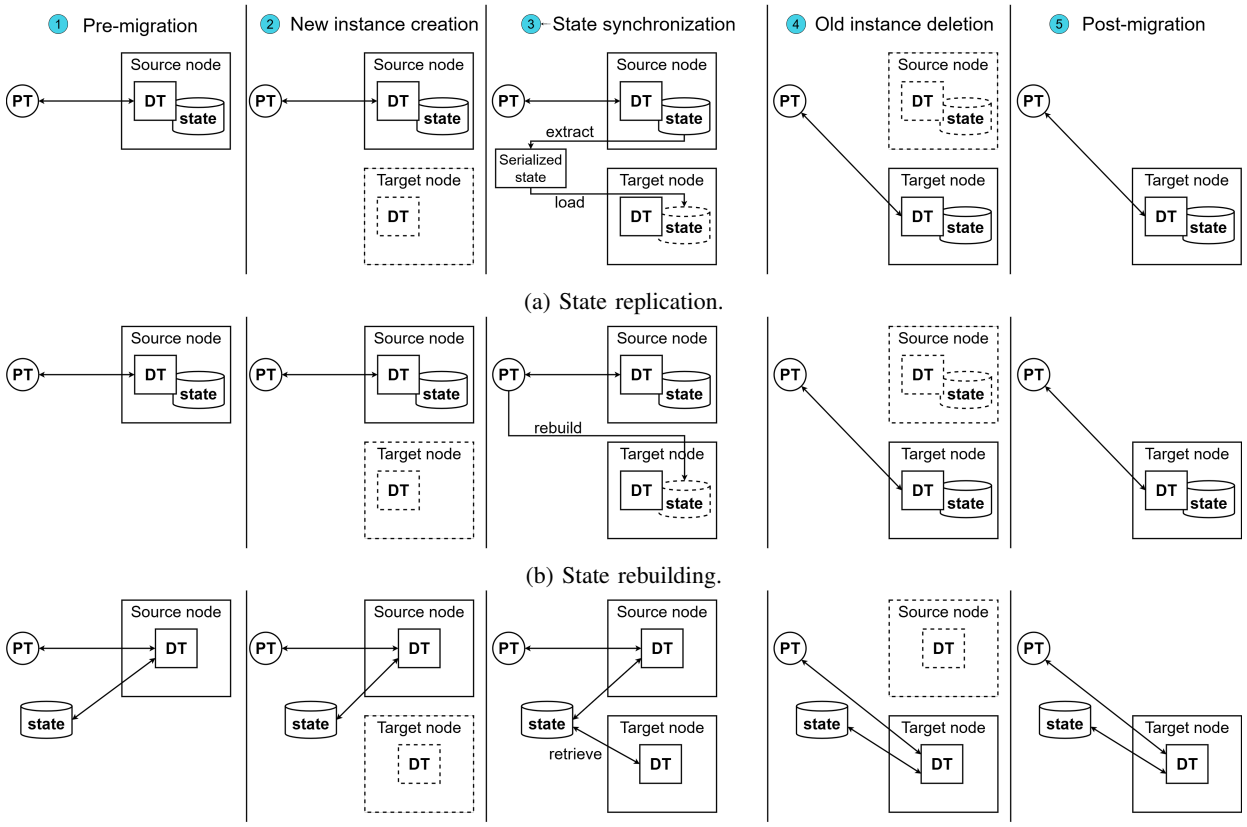
Stateful DT migration is critical from two complementary perspectives—operational efficiency and personnel safety. From the perspective of operational efficiency, incomplete or inconsistent DT state migration can lead to incorrect interpretations of real-time data, potentially resulting in production delays, bottlenecks, or downtime. For example, losing state data regarding conveyor belt speed, machinery configurations, or product progression status could cause inefficient scheduling or resource allocation, thus negatively impacting overall productivity. Even more importantly, from a personnel safety perspective, it is worth noting that critical state information, such as safety interlocks, emergency-stop triggers, or real-time hazard monitoring, directly protects workers from dangerous interactions with equipment. For example, if a robotic arm collision-avoidance data or sensor-triggered emergency conditions are not accurately migrated, safety systems may fail to activate appropriately, increasing the risk of accidents or injuries. Therefore, stateful DT migration is essential not only for preserving operational efficiency but also for ensuring personnel safety in industrial scenarios.

III. State Management Strategies

This section categorizes stateful DTs migration solutions by three state management strategies:

- *State replication*: A state management strategy where the state is copied from the old DT instance to the new one (Section III-A);
- *State rebuilding*: A state management strategy where the new DT instance builds its state from the available data sources (Section III-B);
- *State sharing*: A state management strategy where the old and new DT instances have access to a distributed storage layer containing the state (Section III-C).

Independently from how state migration is performed, migration processes can be split into three phases: (i) creation of a new DT instance, (ii) state synchronization between the new DT instance and the old one, and (iii) deletion of the old DT instance. It is worth pointing out that the sequence of steps that occur throughout a migration process may actually vary. For example, if the state is centrally-managed, the removal of the old DT instance may actually occur before loading the state to the new DT instance.



(a) State replication.

(b) State rebuilding.

(c) Distributed state sharing.

Fig. 1: State management strategies for stateful DT migration.

Each strategy is examined in terms of *downtime*, *total migration time*, *transparency*, and *overhead*. Downtime is the period during which no DT instance is available to gather data or respond to requests; total migration time refers to how long the migration process takes; transparency indicates whether the DT instances are actively involved in the migration process; and resource overhead pertains to the consumption of CPU, memory, and network resources during migration.

A. State Replication

State replication is a state management strategy where the state is copied from one DT instance to another (see Fig. 1a). During the state synchronization phase, this strategy requires two functions: state extraction and state loading (see Fig. 1a-3). These functions can be implemented in different ways. For example, state extraction can be achieved by capturing memory pages and register contents or by explicitly handling variables, functions, classes, and objects. The state loading function must be implemented accordingly. State synchronization may or may not be transparent from the DT point of view, depending on the level of abstraction at which the state is handled. In particular, if state synchronization is managed at the infrastructure level, the entire process may be fully *transparent* to the DT. For example, a state manager module can perform a DT core dump, which can then be used to restore the DT on the target node.

The *total migration time* is highly dependent on the size of the state to be transferred and the available network bandwidth, since the state has to be transmitted from the old to the new DT instance. The *downtime* occurs primarily during state transfer. In this regard, several techniques, such as pre-copy, post-copy, and hybrid approaches, can help optimize this process [25]. In the pre-copy approach, memory pages are copied from source to destination before pausing the old DT instance. In the post-copy approach, the old DT instance is paused first, then minimal state is transferred, and memory pages are fetched on-demand by the new DT instance. The hybrid approach combines pre-copy and post-copy to balance downtime and reliability. However, a downtime is always required to ensure that the transferred state (i.e., the state loaded by the new DT instance) corresponds to the latest computed state (i.e., the state extracted from the old DT instance).

Resource overhead may significantly vary from one implementation of this strategy to another. For instance, a pre-copy approach requires multiple transmission rounds to send memory pages that have changed since the last copy, followed by a final transfer of the remaining state to complete the migration. In contrast, adopting a single bulk state transfer can result in less overhead but potentially higher downtime. Additionally, if the implementation requires synchronous communication between the DT instances, both must remain active

simultaneously for a time period, leading to temporary high resource consumption. A centralized state manager can help mitigate this issue. In fact, such a manager can extract the state from the old DT instance, decommission the old DT instance, activate a new DT instance, and load the extracted state into the new DT instance.

B. State Rebuilding

State rebuilding is a state management strategy where the new DT instance builds its state from the original data sources, i.e., the PT (see Fig. 1b). Note that this strategy can only be used when the state is derivable from the available data sources. The state rebuilding strategy may also be a better design choice than the state replication strategy when building the state is more time-efficient than copying the state. During the state synchronization phase this strategy requires a state rebuilding function (see Fig. 1b-3). This function can be implemented at either the infrastructure or application level. In the former, the migration process is *transparent* to the DT. A buffering mechanism could retain a window of recent physical events, which will be then replayed to the new DT instance. Accordingly, the new DT instance receives the buffered events as if they were live inputs. By doing so, the new DT instance rebuilds the target state without having to explicitly support any state rebuilding logic. In the latter, the DT provides a state rebuilding function that is called at start-up time.

Unlike the state replication strategy (Section III-A), where total migration time depends on state size and bandwidth, the state rebuilding strategy is mainly influenced by the complexity of the rebuilding function. A brief downtime occurs during handoff as the new DT instance takes over.

This state management strategy can ensure an almost seamless migration but may result in high *resource overhead* if high availability is a requirement, as both DT instances must remain simultaneously active. The network traffic generated by the migration process is also typically high due to the requests issued by the DT to build the state. These requests may involve retrieving sensor data, querying historical logs, or synchronizing with external data sources, potentially leading to significant bandwidth consumption.

C. State Sharing

State sharing is a state management strategy where multiple DT instances have access to a distributed shared storage system containing the state (see Fig. 1c). This strategy is particularly suitable in scenarios where a distributed shared storage system is already available and compliant with the DT requirements in terms of latency (i.e., read/write operations do not introduce excessive latency with regard to entanglement) and consistency (i.e., weak, eventual, and strong). This strategy is primarily implemented at the infrastructure level, but may also require application-level support. If the implementation relies on a distributed caching system (see Fig. 1c-3), the DT must provide the functions to read from and write to such a system. This often means that the DT must import a library to interact with the target distributed caching system.

TABLE I: Implementations of the state management strategies.

Strategy	Implementation	Level
State replication	DT API	Infrastructure, Application
	Storage relocation	Infrastructure
State rebuilding	Hot start	Infrastructure
	Cold start	Application
State sharing	Storage rebinding	Infrastructure
	Distributed cache	Infrastructure, Application

Alternatively, the implementation may adopt a distributed shared disk solution, such as a the Network File System (NFS). This implementation is *transparent* to the DT, which perceives the distributed shared storage as part of its local filesystem.

The *total migration time* depends on the time required for the new DT instance to retrieve information from the distributed shared storage system. To minimize *downtime*, the old DT instance can still serve incoming requests until the new DT instance has fully retrieved the necessary data. If so, downtime is primarily determined by the very brief handoff time required by the new DT instance to take over.

Resource overhead is primarily related to the specific distributed shared storage system and the size of the state. In this regard, the determining factors primarily affecting migration are the same as those described for the state replication strategy (see Section III-A)—the size of the state and the available network bandwidth (read operations performed by the new DT instance). However, the state sharing strategy does not necessarily require that the two instances are simultaneously available.

IV. Implementation

This section describes how we implemented the identified state management strategies for stateful DT migration. Specifically, we developed six different implementations, two for each identified strategy (see Table I). We also implemented a proof-of-concept DT of an emulated rotating machine that performs a vibration analysis. The rotating machine, i.e., the PT, publishes a physical event every second. Physical events are published as Message Queuing Telemetry Transport (MQTT) messages (a protocol widely adopted by IIoT applications), unless otherwise specified. The DT maintains a state during execution that consists of the last 100 received physical events (i.e., a 100-second time window). The DT publishes an alert if the average vibration is above a critical threshold.

We deployed the DT as a containerized application in a Kubernetes[1] cluster. In this regard, we extended Kubernetes to support a new type of custom resource—stateful DTs. A custom resource is an extension of the Kubernetes application programming interface (API) that allows to define new Kubernetes objects and define how to handle their lifecycle. Accordingly, we also implemented a Kubernetes operator to provide the logic to manage the lifecycle (i.e., creation, update, and deletion) of stateful DTs, including state management during migration.

[1]https://kubernetes.io/

A. DT API

DT API is an implementation of the *state replication* strategy. In particular, DT API implements the state replication strategy at the application level, with the DT that provides an API for state extraction and loading. When a migration is triggered, the Kubernetes operator extracts the state from the old DT instance and loads it into the new DT instance. To maintain state consistency, the old DT instance disconnects from the MQTT topic before exporting the state, and the new DT instance connects to that MQTT topic after loading the received state. The old DT instance remains active until the new DT instance takes over to minimize downtime.

B. Storage Relocation

Storage relocation is a centrally-managed implementation of the *state replication* strategy. The state extraction and loading functions (de)serialize the state before dumping it in secondary storage. This assumes that a persistent volume is attached to the DT pod (i.e., the minimum deployable unit in Kubernetes). We used an NFS server to provide persistent storage in the Kubernetes cluster. The old and new DT instances access different NFS servers, each with its own storage location. Therefore, a synchronization step is required between the NFS servers. This step is performed using rsync[2] by a sidecar container, which is a container deployed in the same pod as the DT. The sidecar container performs the synchronization before the DT starts. The DT (de)serializes the state, which is encoded in JavaScript Object Notation (JSON). The state extraction and loading functions are called at dismissal and start-up time, respectively, if a state dump file is found. At migration time, the Kubernetes operator requests the deletion of the old DT instance, which, in turn, triggers the extraction of the state. Then, the Kubernetes operator requests the creation of the new DT instance. The sidecar container handles state synchronization between the NFS servers. Finally, at start-up time, the new DT instance finds the state dump file and loads it. As in the DT API implementation, the old DT instance remains active until the new one takes over to minimize downtime.

C. Hot Start

Hot start is an implementation of the *state rebuilding* strategy, where the traffic is duplicated at the infrastructure level for a predefined time window. When migration is triggered, the Kubernetes operator sets up traffic mirroring, leveraging Istio[3] built-in traffic mirroring features. The new DT instance processes the duplicated physical events, while remaining silent. In other words, the new DT instance does not publish anything as long as it is not in charge. Once state synchronization is complete, the Kubernetes operator manages the handoff, transferring control from the old DT instance to the new one. For this implementation, the PT has to directly communicate with the DT instance (i.e., without an MQTT broker) to demonstrate that this approach is feasible even in a direct DT-PT communication scenario. In the case of indirect communication (e.g., with an MQTT broker in between), the implementation would be even more straightforward. In fact, it would be sufficient for the new DT instance to subscribe to the same topic as the old instance, remaining silent until state synchronization is complete.

D. Cold Start

Cold start is an implementation of the *state rebuilding* strategy, where the new DT instance directly queries the PT to build the state. The new DT instance remains in a not ready state until the state is fully rebuilt. In contrast to the *hot start* implementation, which is at the infrastructure level, the cold start implementation is at the application level. When a migration is triggered, the Kubernetes operator deletes the old DT instance, creates a new DT instance, and triggers the state rebuilding function of the new DT instance. Therefore, the old and new DT instances never run at the same time, thus minimizing resource overhead at the expense of downtime. In this case, the downtime is mainly related to the amount of data to be collected and the rate at which such data are produced.

E. Storage Rebinding

Storage rebinding is an implementation of the *state sharing* strategy, where the distributed shared storage system is NFS. Specifically, there is a persistent volume mounted in any DT instance provided by the same NFS server. When migration is triggered, the Kubernetes operator (i) unmounts the persistent volume from the old DT instance, (ii) deletes the old DT instance, (iii) creates the new DT instance, and (iv) mounts such a volume to the new DT instance. As in the *cold start* implementation, the old and new DT instances never run at the same time, thus minimizing resource overhead at the expense of downtime. Here, however, the downtime is mainly related to how long it takes to mount the volume in the new DT instance.

F. Distributed Cache

Distributed cache is an implementation of the *state sharing* strategy, where the state is stored in a Redis[4]-based distributed caching system. Redis is an open source, in-memory, key-value store that can be used as an application cache. The DT instance is implemented to explicitly use the cache and store the state every time it is updated. At start-up time, the new DT instance searches the cache for previously saved state and retrieves it, if any. At migration time, the Kubernetes operator requests the deletion of the old DT instance and, then, creates the new DT instance. Note that, also in this implementation, there is no point in time when both the DT instances are running simultaneously, thus minimizing resource overhead at the expense of downtime, with the determining factor related to state retrieval.

[2]https://rsync.samba.org/

[3]https://istio.io/

[4]https://redis.io/

V. Evaluation

A. Objectives

The evaluation analysis aims to quantitatively assess the state management strategies for stateful DT migration, as discussed in Section III—namely, *state replication*, *state rebuilding*, and *state sharing*. It seeks to answer the following questions for each implementation presented in Section IV:

- Question 1 (Q1). What is the duration of the downtime?
- Question 2 (Q2). What is the total migration time?
- Question 3 (Q3). What is the resource overhead?

B. Metrics

We relied on the following metrics to answer the identified research questions:

- Downtime (Q1): Interval between the last (successful) response from the old instance and the first from the new.
- Total migration time (Q2): Interval between the migration request and the handoff to the new DT instance.
- CPU usage (Q3): Cores consumed during migration.
- Network usage (Q3): Bytes sent/received during migration.

C. Setup

The testbed consisted of a Minikube[5] cluster with 3 nodes. Each node was equipped with 14 CPU cores, 32 GB of memory, and Ubuntu 22.04.5 LTS. We used Docker[6] as container runtime with the Kindnet[7] network plugin. We configured the Kubernetes cluster with the dashboard and metrics-server addons, which provide a web-based user interface for cluster operations and support for metrics collection, respectively. We also deployed cAdvisor[8] to gain more granular control and visibility over container-level resource usage, Istio as service mesh, and Prometheus to automatically scrape the metrics of interest. Lastly, we also installed Mosquitto as MQTT broker to relay MQTT traffic from the PT to the DT, if necessary.

D. Results

We configured the PT, which emulates a vibration sensors, to send a state update every second. When the DT receives a physical state, it computes the average vibration and triggers an alert if the computed value is beyond a certain threshold. During execution, a DT instance builds around 9 MB of state, which mainly consists of the last received 100 physical states.

Table II shows the downtime (Q1) and total migration time (Q2) of each implementation. The experiments were repeated 10 times for each implementation.

DT API was the implementation that achieved the lowest downtime (0.98 s $\pm$ 0.03 s). This is because, in this implementation, the old DT instance remains active until the new instance fully takes over. All implementations, with the exception of *cold start*, exhibited an average downtime of no more than 5 s. Specifically, the downtimes were: 1.93 s $\pm$ 0.33 s (*distributed cache*), 3.15 s $\pm$ 0.46 s (*storage relocation*), 3.17 s $\pm$ 0.34 s (*storage rebinding*), and 4.86 s $\pm$ 5.03 s (*hot start*). As expected, the *cold start* implementation resulted in the highest downtime, namely 103.76 s $\pm$ 0.54 s. This behavior is due to the fact that, in the *cold start* implementation, downtime is directly determined by the amount of data to be collected and the rate at which they are produced. Given that the PT generates 1 physical state per second and the DT must accumulate 100 physical states to build its internal state, the downtime cannot be less than 100 s.

TABLE II: Downtime and total migration time.

Implementation	Downtime [s]	Total Migration Time [s]
DT API	0.98 $\pm$ 0.03	2.13 $\pm$ 0.28
Storage relocation	3.15 $\pm$ 0.46	3.29 $\pm$ 0.46
Hot start	4.86 $\pm$ 5.03	102.22 $\pm$ 0.46
Cold start	103.76 $\pm$ 0.54	103.91 $\pm$ 0.54
Storage rebinding	3.17 $\pm$ 0.34	3.33 $\pm$ 0.34
Distributed cache	1.93 $\pm$ 0.33	2.10 $\pm$ 0.33

The relatively high standard deviation observed in the downtime of the *hot start* implementation (5.03 s) is primarily attributed to the variability introduced by the handoff procedure. This process is orchestrated by the custom Kubernetes operator, which modifies an Istio custom resource to redirect physical event traffic to the new DT instance. The operation exhibits inherent variability, as the modification of the custom resource triggers the Istio controller, which subsequently executes the rerouting logic. The multi-stage nature of this mechanism introduces non-determinism, thereby contributing to the increased variability in downtime.

DT API and *distributed cache* implementations had the shortest total migration times, recording 2.13 s $\pm$ 0.28 s and 2.10 s $\pm$ 0.33 s, respectively. These were followed by *storage relocation* and *storage rebinding*, which reported total migration times of 3.29 s $\pm$ 0.46 s and 3.33 s $\pm$ 0.34 s, respectively. In contrast, the total migration times for the *state rebuilding* strategy implementations—*hot start* and *cold start*—were two orders of magnitude higher. Specifically, the *hot start* implementation required 102.22 s $\pm$ 0.46 s, while the *cold start* implementation took 103.91 s $\pm$ 0.54 s. As previously noted, this significant difference is primarily due to the 100 seconds needed solely to build the DT state.

Figures 2 and 3 show CPU and network overhead, respectively, during migration (Q3). For each implementation, we triggered the DT migration 20 s after the start of the experiment. Each experiment had a total duration of 180 s.

The *distributed cache* implementation was the most resource-intensive, with an average CPU usage of 0.173 $\pm$ 0.004 cores and network usage of 8.242 $\pm$ 0.156 MB during migration. Notably, CPU and network usage remained consistent before, during, and after the migration. On the other end of the spectrum, the *cold start* implementation proved to be the most resource-efficient. Similar to the *distributed cache* implementation, CPU and network usage remained nearly constant throughout the entire experiment, with averages of

[5]https://minikube.sigs.k8s.io/
[6]https://www.docker.com/
[7]https://github.com/aojea/kindnet
[8]https://github.com/google/cadvisor

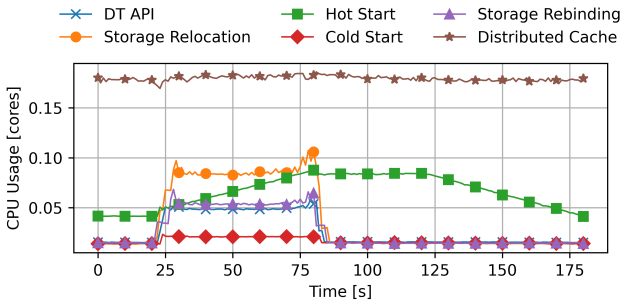


Fig. 2: CPU usage.

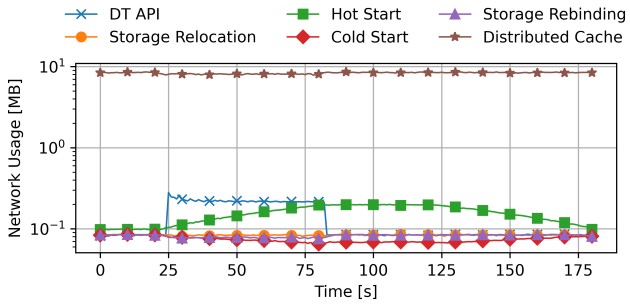


Fig. 3: Network (traffic sent and received) usage.

0.018 ± 0.003 cores and 0.072 ± 0.005 MB, respectively.

The maximum CPU usage observed for the remaining implementations was 0.026 cores for *DT API*, 0.036 cores for *storage rebinding*, 0.088 cores for *hot start*, and 0.048 cores for *storage relocation*. In terms of average CPU usage, the values were 0.018 ± 0.005 cores for *DT API*, 0.028 ± 0.010 cores for *storage rebinding*, 0.074 ± 0.013 cores for *hot start*, and 0.029 ± 0.017 cores for *storage relocation*.

Network usage remained nearly constant for the *storage rebinding* and *storage relocation* implementations, averaging 0.083 ± 0.001 MB and 0.084 ± 0.001 MB, respectively. For the remaining implementations, the maximum observed network usage was 0.085 MB for *DT API* and 0.200 MB for *hot start*. Their corresponding average network usage values were 0.084 ± 0.001 MB and 0.167 ± 0.034 MB, respectively.

In conclusion, the results suggest that no single solution fits all scenarios. The ideal choice depends on the deployment context. Specifically, the *distributed cache* implementation may not be ideal for resource-constrained environments, despite its strong performance in downtime and total migration time. In contrast, the *hot start* implementation shows a gradual rise in CPU usage due to the incremental rebuilding of internal state. Although such behavior may benefit systems that favor ramped resource allocation over abrupt peaks, the extended resource activity still makes it relatively expensive. The DT API implementation continues to stand out for its balanced resource profile, featuring modest CPU and network usage. The *storage rebinding* and *storage relocation* approaches introduce brief CPU usage spikes with minimal network overhead, resulting in a small and localized footprint—an attractive trait for systems with tight resource budgets and tolerance for short service interruptions. Finally, the *cold start* implementation emerges as the most resource-efficient overall, albeit at the cost of high downtime and prolonged total migration time.

VI. Related Work

In the literature, various works have investigated the problem of state management during migration. These works primarily focus on VMs and containerized applications. Although these works offer valuable insights on how to tackle the problem of state management for stateful DT migration, they do not account the cyber-physical nature of DTs. To the best of our knowledge, there is no prior work on state management strategies for stateful DT migration.

A. State Replication

Oh and Kim [12] developed a live container migration approach for stateful containers based on Checkpoint/Restore In Userspace (CRIU)[9]. A checkpoint of the container running on the source host is first created, then this checkpoint is transmitted to the target host, where a new container is started with restored in-memory state based on the checkpoint data.

In the context of fog computing, Puliafito et al. [13] extensively evaluated both cold and live (i.e., pre-copy, post-copy, and hybrid) techniques for stateful container migration. Bellavista et al. [14] designed a framework that both supports application-agnostic and application-aware container migration for edge services. The former, which does not require any application-specific knowledge, copies all the service data to the target host. The latter, instead, targets only a subset of the service data.

B. State Rebuilding

Liu et al. [15] proposed a mechanism for live VM migration as an alternative to pre-copy mechanisms, such as [26] and [27]. This mechanism is based on an event tracing system on the target host and a replay system on the target host. The replay system synchronizes the target host by iteratively replaying the events traced on the source host.

The literature on state rebuilding is limited. Nonetheless, state rebuilding may prove effective in the context of DTs, where the PT serves as a continuous source of physical events (i.e., a form of trace data). This is particularly advantageous in scenarios where the DT state can be entirely derived from these physical events. In such cases, a dedicated tracing system at the DT level may not even be required. Instead, it is sufficient to replay (i.e., duplicate) the physical events until the DT has accumulated enough information to build the state.

C. State Sharing

Nadgowda et al. [16] implemented Voyager, a framework designed for live container migration. Voyager provides i) in-memory state migration leveraging CRIU, ii) local filesystem

[9]https://criu.org/

migration, which combines data federation (making source host data accessible to the target host without immediately copying data) and lazy replication (transferring data asynchronously in the background), and iii) network filesystem migration. Voyager relies on both state sharing (data federation) and state replication (lazy replication). However, we chose to categorize it here primarily due to its distinctive use of data federation for state sharing.

State sharing is also supported natively by Kubernetes—the de facto industry standard container orchestration system—with the mechanisms of persistent volumes and persistent volume claims, which enable pods to persist data across restarts and migrations. A persistent volume is provided by a storage provider and can be mounted on multiple pods simultaneously.

VII. Conclusion

In this work, we have identified three state management strategies for stateful DT migration—*state replication*, *state rebuilding*, and *state sharing*. Then, we have developed six implementations, two for each strategy. In particular, the *DT API* and *storage relocation* implementations for the *state replication* strategy; the *hot start* and *cold start* implementations for the *state rebuilding* strategy; and the *storage rebinding* and *distributed cache* implementations for the *state sharing* strategy. Each implementation has been experimentally evaluated in an industrial setting. The results show that no single solution fits all scenarios; instead, each involves trade-offs in terms of downtime, total migration time, and resource overhead.

Acknowledgment

This work was partially supported by the Italian Ministry of Education and Research (MUR) PRIN 2022 PNRR DATRUST Project (Project ID: P20225KTR4, CUP: I53D23006060001) and by the European Union - NextGenerationEU, PNRR SERICS PE00000014 Spoke 8 Cascading Funding Z4I Project (CUP 33C22002810001).

References

[1] M. W. Grieves, *Digital Twins: Past, Present, and Future*, 2023, vol. 1.

[2] Y. Fang, C. Peng, P. Lou, Z. Zhou, J. Hu, and J. Yan, "Digital-twin-based job shop scheduling toward smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 12, pp. 6425–6435, 2019.

[3] A. Castellani, S. Schmitt, and S. Squartini, "Real-world anomaly detection by using digital twin systems and weakly supervised learning," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 7, pp. 4733–4742, 2021.

[4] Z. Lv, J. Guo, and H. Lv, "Safety poka yoke in zero-defect manufacturing based on digital twins," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1176–1184, 2023.

[5] K. Josifovska, E. Yigitbas, and G. Engels, "Reference framework for digital twins within cyber-physical systems," in *Proceedings of the 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems*, ser. SEsCPS '19. IEEE Press, 2019, p. 25–31. [Online]. Available: https://doi.org/10.1109/SEsCPS.2019.00012

[6] K. M. Alam and A. El Saddik, "C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE Access*, vol. 5, pp. 2050–2062, 2017.

[7] K. Hribernik, G. Cabri, F. Mandreoli, and G. Mentzas, "Autonomous, context-aware, adaptive digital twins—state of the art and roadmap," *Computers in Industry*, vol. 133, p. 103508, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361521001159

[8] F. Siqueira and J. G. Davis, "Service computing for industry 4.0: State of the art, challenges, and research opportunities," *ACM Comput. Surv.*, vol. 54, no. 9, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3478680

[9] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Requirements and design patterns for adaptive, autonomous, and context-aware digital twins in industry 4.0 digital factories," *Comput. Ind.*, vol. 149, no. C, Aug. 2023. [Online]. Available: https://doi.org/10.1016/j.compind.2023.103918

[10] ——, "An entanglement-aware middleware for digital twins," *ACM Trans. Internet Things*, vol. 5, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3699520

[11] ——, "Exploiting microservices and serverless for digital twins in the cloud-to-edge continuum," *Future Generation Computer Systems*, vol. 157, p. 275 – 287, 2024.

[12] S. Oh and J. Kim, "Stateful container migration employing checkpoint-based restoration for orchestrated container clusters," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, pp. 25–30.

[13] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/7/1488

[14] P. Bellavista, A. Corradi, L. Foschini, and D. Scotece, "Differentiated service/data migration for edge services leveraging container characteristics," *IEEE Access*, vol. 7, pp. 139 746–139 758, 2019.

[15] H. Liu, H. Jin, X. Liao, C. Yu, and C.-Z. Xu, "Live virtual machine migration via asynchronous replication and state synchronization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 1986–1999, 2011.

[16] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2137–2142.

[17] M. Grieves, "Completing the cycle: Using plm information in the sales and service functions [slides]," in *SME Management Forum*, 2002.

[18] J. B. Heluany and V. Gkioulos, "Survey on digital twins: from concepts to applications," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. New York, NY, USA: Association for Computing Machinery, 2023.

[19] K. Y. H. Lim, P. Zheng, and C.-H. Chen, "A state-of-the-art survey of digital twin: techniques, engineering product lifecycle management and business innovation perspectives," *Journal of Intelligent Manufacturing*, vol. 31, no. 6, p. 1313 – 1337, 2020.

[20] Y. Wu, K. Zhang, and Y. Zhang, "Digital twin networks: A survey," *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13 789–13 804, 2021.

[21] B. R. Barricelli, E. Casiraghi, and D. Fogli, "A survey on digital twin: Definitions, characteristics, applications, and design implications," *IEEE Access*, vol. 7, pp. 167 653–167 671, 2019.

[22] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the iot context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020.

[23] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Odte: A metric for digital twin entanglement," *IEEE Open Journal of the Communications Society*, vol. 5, pp. 2377–2390, 2024.

[24] M. Azarmipour, H. Elfaham, C. Gries, T. Kleinert, and U. Epple, "A service-based architecture for the interaction of control and mes systems in industry 4.0 environment," in *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*, vol. 1, 2020, pp. 217–222.

[25] K. Kaur, F. Guillemin, and F. Sailhan, "Container placement and migration strategies for cloud, fog, and edge data centers: A survey," *International Journal of Network Management*, vol. 32, no. 6, 2022.

[26] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," 2005, p. 273 – 286.

[27] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," ser. ATEC '05. USA: USENIX Association, 2005, p. 25.