UDC 004.4 DOI https://doi.org/10.35546/kntu2078-4481.2025.1.2.8

A. M. HUBSKYI
Ph.D., Associate Professor at the Department of Computer Science
and Software Engineering
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
ORCID: 0000-0002-5361-5485

YA. I. KORNAHA
Doctor of Technical Sciences, Professor,
Dean of the Faculty of Computer Science and Computer Engineering
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
ORCID: 0000-0001-9768-2615

O. I. MARCHENKO
Senior Lecturer at the Department of Computer Science and Software Engineering
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
ORCID: 0000-0001-5754-4920

DYNAMIC RESILIENCE MECHANISM FOR SCALABLE INFORMATION INFRASTRUCTURES

This article examines the challenge of ensuring the stability of computing systems under peak loads. The main problem is that traditional monitoring methods often fail to detect critical failures and unforeseen situations on time, which may lead to data loss and significant economic damage. The research aims to develop a hybrid monitoring architecture that combines synchronous polling of critical parameters with asynchronous event-driven data collection. To achieve this goal, the proposed solution employs virtual probes that allow for system analysis without significantly impacting performance and a component capable of detecting anomalous system states.

Experimental studies have confirmed the effectiveness of the proposed model, which reduces the risk of failures, optimizes the use of computing resources, and ensures high system scalability under various load conditions. The proposed model was tested on real-world scenarios using simulation environments that emulate emergency situations and intensive query streams. The research results demonstrate a significant improvement in system efficiency, a reduction in response time to failures, and an optimization of information systems' performance. The obtained data also indicates the possibility of integrating the proposed approach with existing solutions for monitoring and managing computing systems.

Considerable attention was paid to analyzing the impact of various monitoring parameters on performance, which allowed the determination of optimal values for balancing data collection accuracy and system load. A comparative analysis with traditional monitoring methods was also conducted, revealing that the hybrid approach provides a more stable system operation during peak loads. The summarized research findings may serve as a foundation for further scientific investigations and implementing innovative technologies in computing resource management.

Key words: *heavy loads, computing resource scalability, monitoring, microservice architecture, information systems, design patterns, hybrid architecture.*

А. М. ГУБСЬКИЙ
кандидат технічних наук,
доцент кафедри інформатики та програмної інженерії
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
ORCID: 0000-0002-5361-5485

Я. І. КОРНАГА
доктор технічних наук, професор,
декан факультету інформатики та обчислювальної техніки
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
ORCID: 0000-0001-9768-2615

О. І. МАРЧЕНКО
старший викладач кафедри інформатики та програмної інженерії
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
ORCID: 0000-0001-5754-4920

ДИНАМІЧНИЙ МЕХАНІЗМ СТІЙКОСТІ ДЛЯ МАСШТАБОВАНИХ ІНФОРМАЦІЙНИХ ІНФРАСТРУКТУР

Ця стаття аналізує проблему забезпечення стабільності обчислювальних систем під час пікових навантажень. Основна проблема полягає в тому, що традиційні методи моніторингу часто не встигають виявити критичні збої та непередбачені ситуації, що може призвести до втрати даних та значних економічних збитків. Дослідження має на меті розробити гібридну архітектуру моніторингу, що поєднує синхронне опитування критичних параметрів із асинхронним збором даних на основі подій. Для досягнення цієї мети запропоноване рішення використовує віртуальні датчики, які дозволяють аналізувати систему без значного впливу на її продуктивність, а також компонент який, здатний розрізняти аномальні стани системи.

Експериментальні дослідження підтвердили ефективність запропонованої моделі, яка знижує ризик збоїв, оптимізує використання обчислювальних ресурсів та забезпечує високу масштабованість системи за різних умов навантаження. Запропонована модель була протестована на реальних сценаріях із використанням симуляційних середовищ, що імітують аварійні ситуації та інтенсивні потоки запитів. Результати дослідження демонструють значне покращення ефективності системи, скорочення часу реагування на збої та оптимізацію продуктивності інформаційних систем. Отримані дані також свідчать про можливість інтеграції запропонованого підходу з існуючими рішеннями для моніторингу та управління обчислювальними системами.

Значну увагу було приділено аналізу впливу різних параметрів моніторингу на продуктивність, що дозволило визначити оптимальні значення для збалансування точності збору даних та навантаження на систему. Було проведено порівняльний аналіз з традиційними методами моніторингу, який виявив, що гібридний підхід забезпечує більш стабільну роботу системи під час пікових навантажень. Узагальнені результати дослідження можуть стати основою для подальших наукових досліджень та впровадження інноваційних технологій в управлінні обчислювальними ресурсами.

Ключові слова: *пікові навантаження, масштабованість обчислювальних ресурсів, моніторинг, архітектура мікросервісів, інформаційні системи, шаблони проектування, гібридна архітектура.*

Problem statement

Modern information systems are complex systems constituted by intricate architectures and diverse couplings. When the system is under peak load, some modules may fail in the processing mode or enter an unstable state, resulting in overall system instability and making it impossible to process all incoming requests correctly. System resilience is one of the most important challenges, especially in continuous processing.

There are several strategies for reducing these risks. In some situations, a system can activate a «safe mode,» providing a suboptimal alternative for a more significant number of requests and, in a different approach, scaling the computational resources by adding more dedicated modules to process incoming requests. However, the trick is knowing when and how to do so successfully.

One of the important tasks is reconciling the level of service provided with the necessity to avoid catastrophic failure. Traditional monitoring approaches are based on static thresholds that cannot catch the variability and evolution of system workloads. Furthermore, reactive methodologies that recover only after the performance degradation has already materialized may be too late, risking cascading failures. The existing generic and top-down approaches to resilience are not enough, and more adjustable and preemptive resilience mechanisms that can evaluate a system's health and implement stability measures at runtime are urgently needed.

Analysis of the latest research and publications

The literature on self-adaptive systems focuses on the need for real-time decision-making to keep the system steady and responsive to workload changes. The insufficiency of conventional, static threshold-based monitoring methods has been noted in related dependability and fault tolerance studies. These approaches are often ill-equipped to manage complex interactions among system components and are slow to respond when failures occur.

A few works recommend proactive countermeasures based on continuous system monitoring and a predictive analytics approach. One such means is to seamlessly interconnect explicit performance indicators (CPU utilization, memory usage, response times) to implicit signals indicating impending failure long before the problem manifests into a disaster. Research indicates that machine learning techniques and anomaly detection models help improve failure prediction and optimize resource usage.

Furthermore, the latest updates in the virtualization environment involve implementing advanced monitoring systems such as virtual probes and virtual spectators. These allow for real-time observation of system behavior without incurring

the performance cost commonly associated with traditional monitoring tools [1]. By providing insightful data and automation, such mechanisms can also be designed to enhance resilience against unexpected failures.

Building upon this research, we propose a dynamic resilience framework that leverages virtual probes and a virtual spectator to enhance system monitoring and stability significantly. Each process is adapted from previously proven methods, utilizing an aggregated, real-time analysis of system health indicators over a prolonged period. This approach alleviates the computational burden of high-frequency monitoring activities while ensuring improved system resilience.

Formulation of the research objective

System owners can gain unprecedented visibility into system operations by deploying virtual probes at strategic points throughout the system architecture and employing a virtual spectator to aggregate and analyze the collected data. This enhanced observability enables more nuanced decision-making regarding when to activate safe mode protocols or initiate resource scaling operations, ultimately leading to more stable and resilient system performance even under extreme load conditions [2].

Core Findings and Analysis

To implement such an observability framework, the system relies on probes that measure key system metrics and report them to an observer component. As shown in Figure 1, these probes monitor various aspects of system health, such as active requests, memory usage, and cache entries, allowing for real-time assessment of the system state.

Each probe in the system measures a specific metric, such as active requests, memory usage, or cache entries, to determine the system state.

The result of a probe's check can be either:

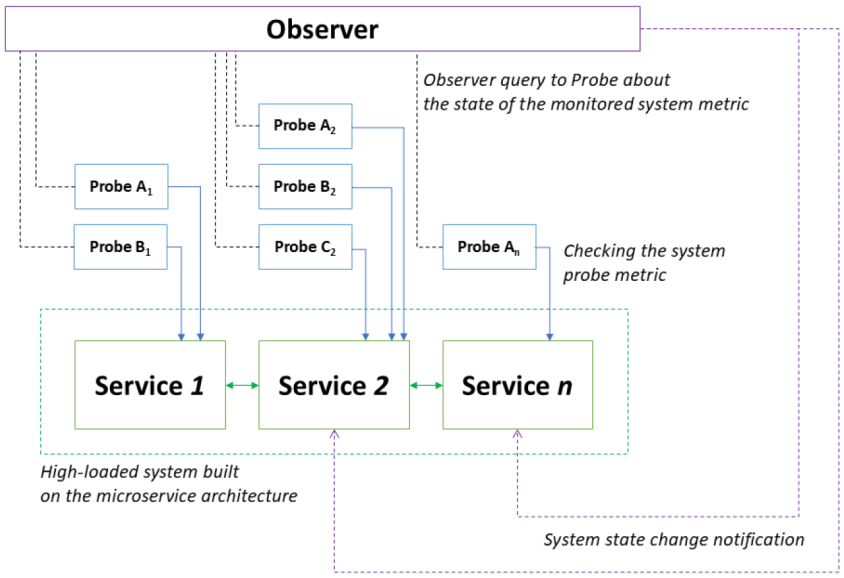

Fig. 1. Example of the interaction between a high-load information system, its Probes, and the Spectator

- **true:** Indicating that the metric is within normal limits.
- **false:** Indicating that the metric has deviated from normal operating conditions.

$$F_{probe_i}(T) = [0,1]. \tag{1}$$

Each Probe is associated with one or more **spectators**, and each **spectator** may manage multiple probes. A **spectator** must implement the *ISpectator* interface (see Figure 2 in the original document) and generate events when changes occur or after probe polling.

$$J(T) = \sum_{i=1}^{n} F_{probe_i}(T). \tag{2}$$

During a periodic check, the **spectator** queries all its probes, forming an array that is then recorded in the state journal (denoted as J(T) in formula (2), where T is the time of polling). Based on these journal entries, the spectator computes the current system state, S(T) (as described by formula (3)).

$$S(T) = F(J(T_1), J(T_2), \ldots, J(T_m)). \tag{3}$$

If a change is detected – i. e., if $S(T_{(i-1)}) \neq S(T_i)$ – the **spectator** triggers an event. Modules that subscribe to this event can then adjust their parameters accordingly.

```
7   public interface ISpectator<TState> where TState : struct, IConvertible
8   {
9       /// Event that is triggered when the state of the spectator changes. ...
        Andrew Gubskiy
12      event EventHandler<StateEventArgs<TState>> StateChanged;
13
14      /// Event that is triggered when the health of the spectator is checked. ...
        Andrew Gubskiy
17      event EventHandler<HealthCheckEventArgs> HealthChecked;
18
19      /// State of the spectator. ...
        1 implementation  Andrew Gubskiy
22      TState State { get; }
23
24      /// Uptime of the spectator. ...
        1 implementation  Andrew Gubskiy
27      TimeSpan Uptime { get; }
28
29      /// Name of the spectator. ...
        1 implementation  Andrew Gubskiy
32      string Name { get; }
33
34      /// Adds a probe to the spectator. ...
        1 implementation  Andrew Gubskiy
38      void AddProbe(IProbe probe);
39
40      /// Checks the health of the spectator. ...
        1 implementation  Andrew Gubskiy
43      void CheckHealth();
44  }
```

Fig. 2. ISpectator interface

The system state may be evaluated using various methods:

- **Boolean Evaluation:** The system is either operational (true) or non-operational (false).
- **Enumerated Evaluation:** The system can be in one of several defined states (e. g., normal, warning, danger, failure).
- **Discrete Evaluation:** The state is expressed numerically (from 0 to 1000, where 1000 indicates full operability and zero complete failure).
- **Percentage Evaluation:** The system's operability is expressed as a percentage (with 100 % representing full operability and 0 % representing complete failure).

Beyond these, any class implementing the IStateEvaluator interface can be used for state calculation. This approach is potent for languages like C#, Java, and others when combined with the inversion of control (IoC) principle [3].

Probes can be integrated via dependency injection using an IoC container. At the same time, spectators can be implemented as singletons – ensuring that a single instance is accessible to various components or services within the module.

```
7   [PublicAPI]
    12 usages  4 inheritors  Andrew Gubskiy +1
8   public interface IStateEvaluator<TState>
9   {
10      /// <summary>
11      /// Evaluates the state of the spectator.
12      /// </summary>
13      /// <param name="currentState">
14      /// Current state of the spectator.
15      /// </param>
16      /// <param name="stateChangedLastTime">
17      /// The time when the state of the spectator was changed last time.
18      /// </param>
19      /// <param name="journal">
20      /// Journal of the spectator.
21      /// </param>
22      /// <returns></returns>
        2 usages  4 implementations  Andrew Gubskiy +1
23      TState Evaluate(TState currentState, DateTime stateChangedLastTime, IReadOnlyCollection<JournalRecord> journal);
24  }
```

Fig. 3. IStateEvaluator interface

The proposed approach leverages a framework that embeds components for tracking state changes into an existing information system with minimal code modification. Although the article demonstrates an implementation using the. NET platform and the C# programming language, the approach is adaptable to any object-oriented language and many functional programming languages.

This framework operates through a layer that intercepts key system events and state transitions without requiring extensive refactoring of the existing codebase. The instrumentation is achieved through aspect-oriented programming techniques and lightweight proxies that wrap critical system components [4]. These wrappers expose standardized interfaces for the monitoring subsystem while maintaining the original component behaviors, ensuring system functionality remains unaffected during regular operation.

The implementation showcased in the article utilizes.NET's reflection capabilities and dynamic proxy generation to create these monitoring hooks at runtime. This approach allows for selective instrumentation of only those components deemed critical for system stability assessment, minimizing the performance overhead.

Components of the Monitoring System

- **Probe:** Responsible for checking the state of a specific system metric.
- **Spectator:** Monitors one or several probes and aggregates their outputs.
- **Journal:** A record (or array) of probe readings, each marked with the time of the check. The **Spectator** maintains it.
- **State Evaluator:** Calculates the system's overall state based on journal entries. The **Spectator** can use different implementations of the **State Evaluator**.

The **Spectator** generates two types of events: one when it has queried all probes and another when the State Evaluator updates its assessment of the system's state. The **Spectator** and the **Probes** can operate synchronously or asynchronously relative to the system's main processes. When a state change is detected, the **Spectator** emits an event to which one or more system modules may subscribe.

Polling Scenarios

There are two primary scenarios for spectator-initiated probe polling:

- **Asynchronous Polling:** Probes are polled independently of the main system processes, for example, by a timer running on a separate thread at regular intervals.
- **Synchronous Polling:** A probe is polled directly from a module's action, with the subsequent system state recalculated immediately.

Overhead Analysis

The monitoring framework inherently introduces an essential and manageable computational overhead. This study examines the key performance considerations associated with probe polling within the system.

The computational impact of probe polling is principally determined by the polling methodology employed. In the case of synchronous polling, there is a direct impact on response time, as immediate feedback is provided at the expense of potential delays in processing. Conversely, asynchronous polling operates as a background process, thereby diminishing its immediate impact on primary operations; however, it necessitates the allocation of additional CPU and memory resources [5]. The overall computational cost of the probing mechanism is influenced by the number of probes deployed and the frequency of their evaluations. Moreover, the implementation of event throttling serves as a critical control mechanism to prevent excessive processing. It is imperative to recognize that enhanced observability, achieved through integration with real-time data collection and visualization tools, significantly augments system monitoring capabilities.

Infrastructure Integration

The resilience framework seamlessly integrates with modern observability tools to enhance monitoring capabilities. For logging and metrics collection, Prometheus scrapes probe data and visualizes trends, while the ELK Stack (Elasticsearch and Kibana) handles comprehensive events and state change logging. OpenTelemetry provides standardized tracing and metrics collection across the system. Distributed tracing is managed through tools like Jaeger and Zipkin, which track state changes across microservices, complemented by cloud-native solutions such as Azure Monitor and AWS CloudWatch for anomaly detection and alerting.

The framework can extend beyond passive monitoring with alerting and auto-remediation features: if the solution is integrated with Grafana Alerts it can send notifications about threshold breaches, while Kubernetes Horizontal Pod Autoscaler (HPA) dynamically scales resources in response to changing system conditions, creating a self-healing infrastructure that maintains stability under varying loads.

Future Perspectives

The framework discussed here provides a robust foundation for self-diagnosing high-load information systems. Its modular design, which separates the monitoring components (probes, spectator, journal, and evaluator), facilitates maintenance and scalability and allows for the integration of advanced analytics [6]. For instance, incorporating machine learning techniques for anomaly detection could enable the system to predict potential failures before they occur. Adaptive thresholds and real-time data analytics enhance the system's responsiveness to emerging issues.

From a practical standpoint, implementing such a framework requires careful consideration of performance overhead and security, particularly in environments where data integrity is critical. Future research could explore optimized data aggregation strategies and the application of edge computing to distribute monitoring tasks more efficiently [7]. Moreover, integrating these diagnostic mechanisms with automated remediation strategies could transform reactive systems into proactive, self-healing infrastructures.

Conclusions

The proposed approach enables dynamic adjustment of system parameters, allowing the system to maintain correct operation even under peak load conditions. This approach has proven particularly effective in distributed systems built on microservice architecture.

In summary, the self-diagnostic mechanism and its potential enhancements represent a significant advancement in maintaining the reliability and scalability of high-load information systems. By leveraging modern design patterns, robust programming paradigms, and potential future innovations in machine learning and distributed computing, developers can build systems capable of detecting, anticipating, and resolving issues before they impact overall performance.

References

1. NETSCOUT. (n.d.). Virtual Network Probes for Service Assurance in Virtualized Networks. NETSCOUT. Retrieved from https://www.netscout.com/blog/virtual-network-probes

2. Alessi, F., Tundo, A., Mobilio, M., Riganelli, O., & Mariani, L. (2024). ReProbe: An Architecture for Reconfigurable and Adaptive Probes. arXiv preprint arXiv:2403.12703.

3. Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection Pattern. Retrieved from https://martinfowler.com/articles/injection.html

4. Laddad, R. (2009). AspectJ in Action: Enterprise AOP with Spring (2nd ed.). Manning Publications.

5. Yang, J., Minturn, D. B., & Hady, F. (2012). When Poll is Better than Interrupt. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12) (pp. 1–?). USENIX Association. Retrieved from https://www.usenix.org/system/files/conference/fast12/yang.pdf

6. Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2009). Detecting Large-Scale System Problems by Mining Console Logs. In Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07) (pp. 19–19). USENIX Association.

7. Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal, 3*(5), 637–646. DOI: https://doi.org/10.1109/JIOT.2016.2579198