Locating Inter-communication Performance Anti-patterns in Microservices

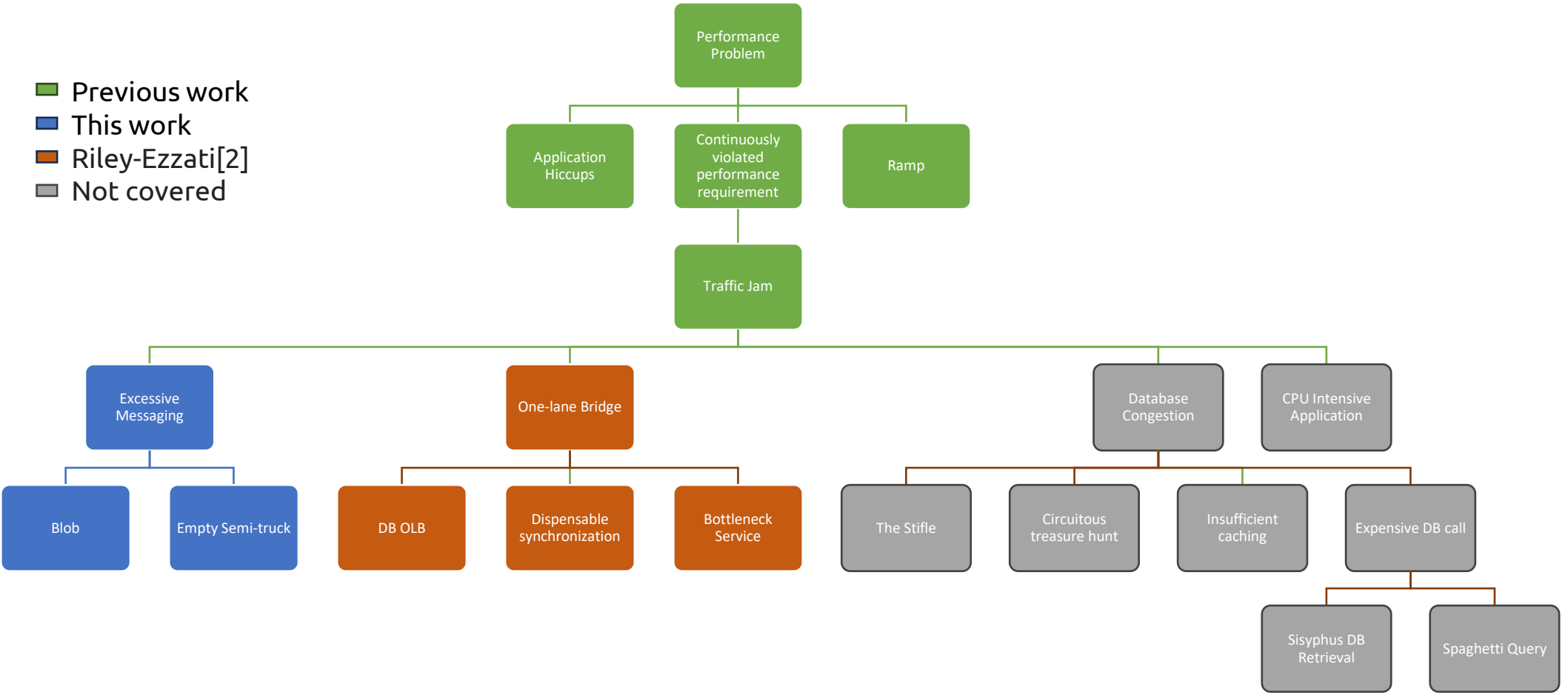
Masoumeh Nourollahi

Polytechnique Montréal

DORSAL Laboratory

Introduction

- Performance anti-patterns (SPAs)
  - Bad practices which result in performance degradation
  - May not cause system failure, but impact performance

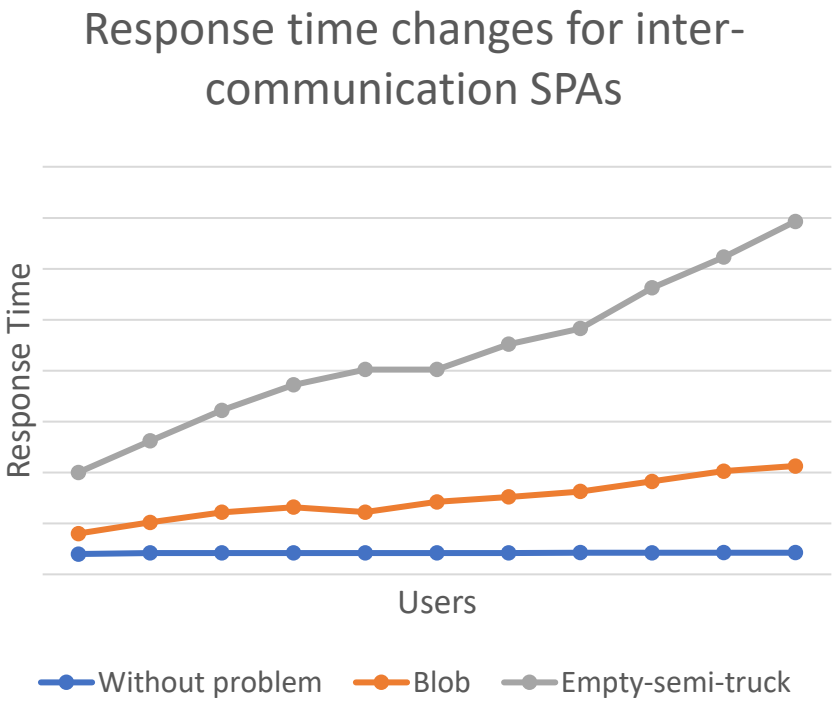


Agenda

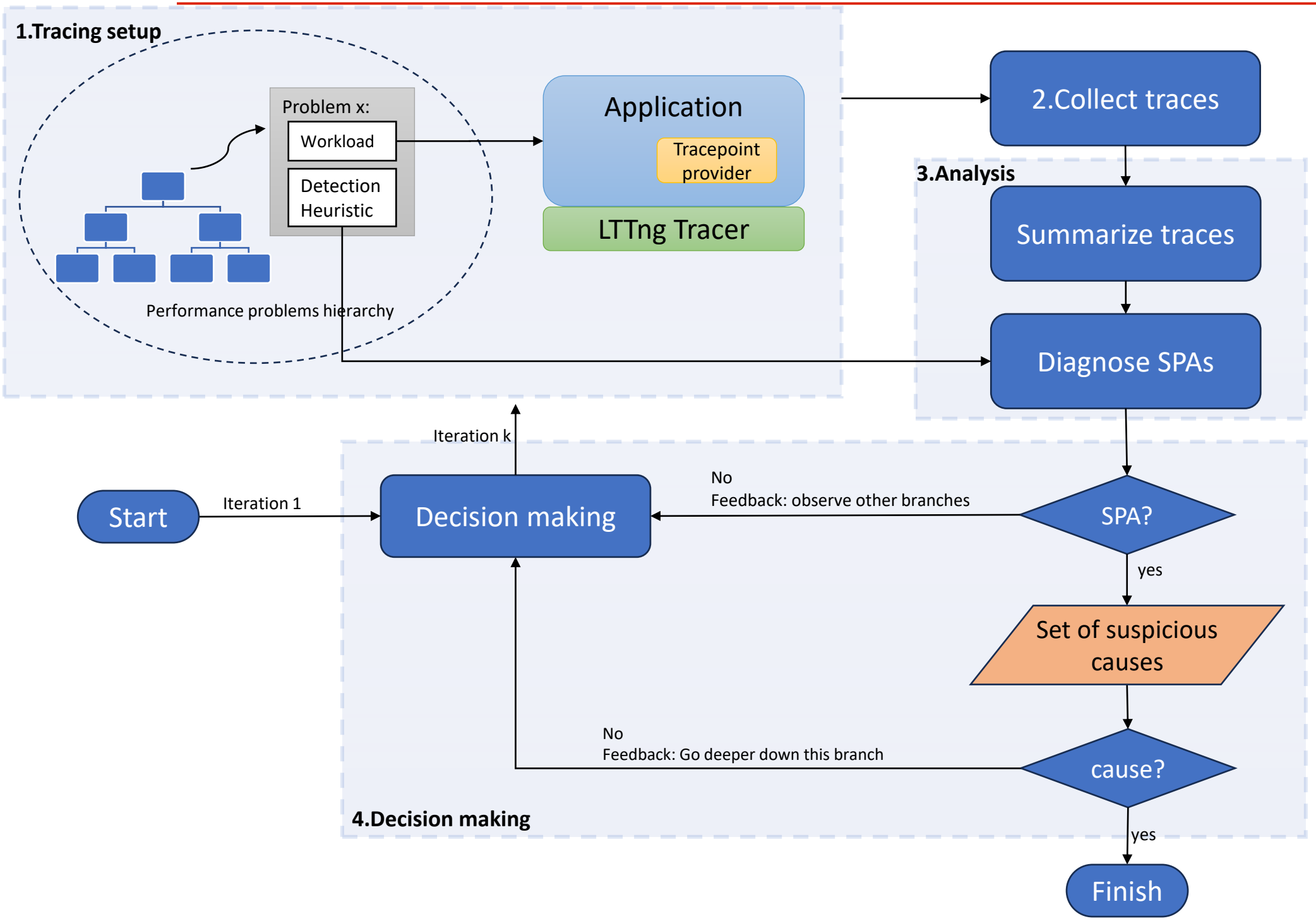
- Problem statement
- Proposed artichecture and detection method
- Experiments
- Results
- Discussion and future work

Problem statement

- Excessive Messaging
- Blob
  - High communication overhead through remote communication between central blob component and other components
- Empty Semi-truck
  - Large number of small messages transmitted between two components as part of a single user request



Proposed Architecture



SPA Detection method

1. Decision making on the required tracing setup (load test and tracing scope)
2. Trace collection
3. Data Aggregation: The data is aggregated over fixed time windows
4. Using heuristics to detect SPA

LOAD PROFILES

1.single user- single action

2.single user- mixed action

3.increasing load- single action

4.increasing load- mixed action

5.max load- single action

6.max load- mixed action

TRACING SCOPE

1.Kernel space

- Kernel events
  - net_'*',sock_'*',napi_poll,skb_'*'
  - Sched_'*'

2.User space

- Function call entry/exit

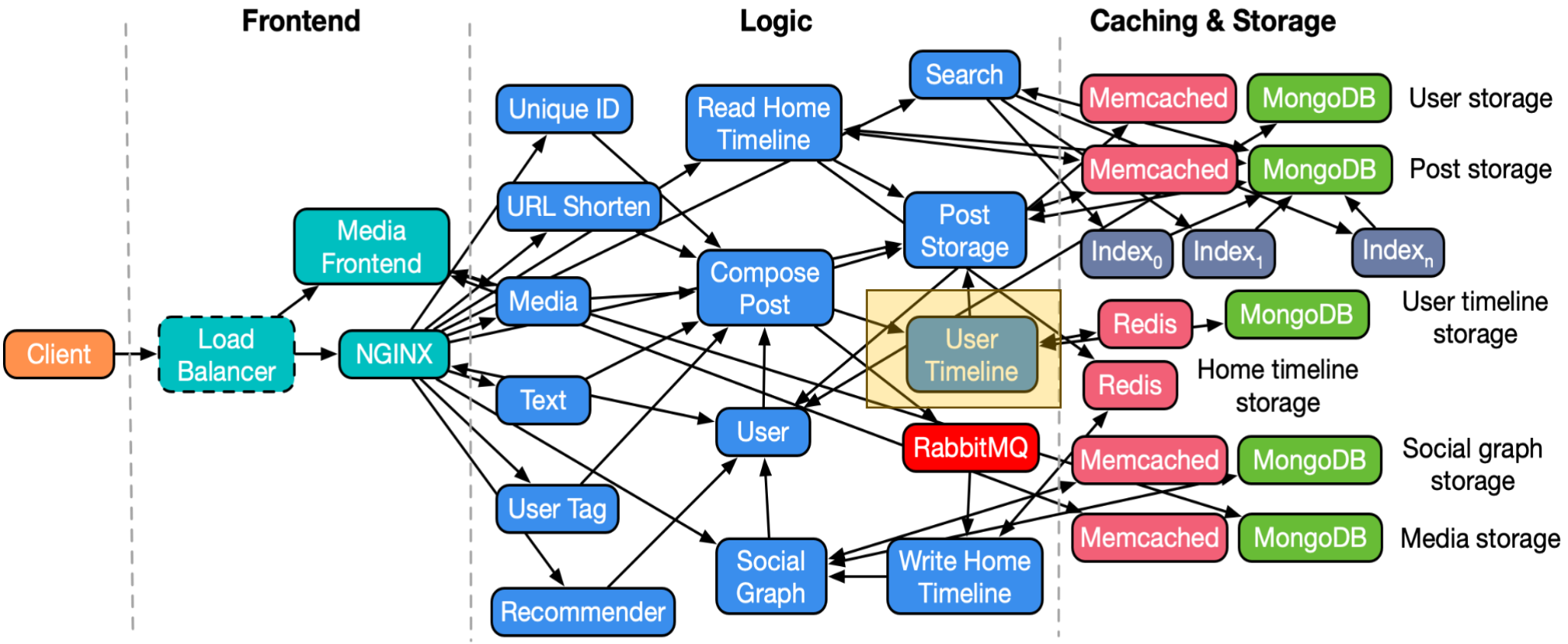
Intercommunication SPAs- Detection Method

- Excessive Messaging
  - Load profile: Increasing load- single action
  - Tracing scope: Kernel events: net_if_receive_skb, net_if_receive_skb
- Blob
  - Load profile: stress load- single action
  - Tracing scope: Kernel events: net_if_receive_skb, net_if_receive_skb
- Empty Semi-truck
  - Load profile: Single user-single action
  - Tracing scope: Kernel events: net_if_receive_skb, net_if_receive_skb, Sched_*, 2 UST Tracepoints- action related method entry/exit

Tracepoint	Definition
Net_dev_queue	A network packet is sent
Net_if_receive_Skb	A network packet is received
Sched_switch	Transition from one task to another on a CPU

Experiments

- TestBed:
  - DeathStarBench- Social Network application



SPA Injection- Excessive messaging

- A loop that performs 1000 consecutive queries to MongoDB. This excessive messaging significantly increases the load on the MongoDB server during read operations.

Original

```
int mongo_start = start + post_ids.size();
std::multimap<std::string, std::string> redis_update_map;
if (mongo_start < stop) {
  // Instead find post_ids from mongodb
  mongoc_client_t *mongodb_client = mongoc_client_pool_pop(
      _mongodb_client_pool);
  if (!mongodb_client) {
    ServiceException se;
    se.errorCode = ErrorCode::SE_MONGODB_ERROR;
    se.message = "Failed to pop a client from MongoDB pool";
    throw se;
  }
  auto collection = mongoc_client_get_collection(
      mongodb_client, "user-timeline", "user-timeline");
  if (!collection) {
    ServiceException se;
    se.errorCode = ErrorCode::SE_MONGODB_ERROR;
    se.message = "Failed to create collection user-timeline from MongoDB";
    throw se;
  }
```

SPA Injected

```
int mongo_start = start + post_ids.size();
std::multimap<std::string, std::string> redis_update_map;
if (mongo_start < stop) {
  // Excessive messaging: Multiple queries to MongoDB
  for (int i = 0; i < 1000; ++i) {
    mongoc_client_t *mongodb_client = mongoc_client_pool_pop(
        _mongodb_client_pool);
    if (!mongodb_client) {
      ServiceException se;
      se.errorCode = ErrorCode::SE_MONGODB_ERROR;
      se.message = "Failed to pop a client from MongoDB pool";
      throw se;
    }
    auto collection = mongoc_client_get_collection(
        mongodb_client, "user-timeline", "user-timeline");
    if (!collection) {
      ServiceException se;
      se.errorCode = ErrorCode::SE_MONGODB_ERROR;
      se.message = "Failed to create collection user-timeline from MongoDB";
      throw se;
    }
```

Results

- Identification of intercommunication performance anti-patterns by kernel traces
- Reduced pre-tracing effort
- Low-cost tracing (2 to 6 kernel events required to identify the SPAs)

Metric	Trubiani (2023) [1]	Avritzer (2002) [6]	Wert (2014) [10]	Our method
Pre-tracing effort	UST instrumentation	UST instrumentation	UST instrumentation	1 UST TP
Tracing cost/overhead	Not reported	Not reported	High for EST pattern	Between 2 to 6 kernel events
Detection method	Load testing and profiling	Queuing performance models	Algorithmic approach	Statistical analysis and Algorithmic approach
Accuracy	F-measure > 85%	F-measure > 44%	True positive ≈ 1 Notable False positive	True positive ≈ 1 Notable False positive

Discussion

1. Considering time-window analysis instead of per-request

2. Previous works all used UST traces to detect intercommunication anti-patterns, we are mostly using kernel traces
   - Less effort pre tracing (does not need any or very little instrumentation)
   - Avoiding unknown impact on system performance
   - Taking 2 to 6 kernel events which is very low-overhead tracing

3. We implemented a number of methods to compare
   - Wert statistical heuristics to detect performance anti-patterns [10]
   - Avritzer queuing model for anti-pattern detection [6]

Discussion

- Future directions:
  - Metrics based comparison of application performance to detect SPAs
  - Use of time-series analysis methods by using the available metrics

- Some suggested metrics to measure in each time window
  - msg_service_time
  - request_throughput
  - message_throughput
  - request_arrival_rate
  - msg_arrival_rate
  - msg_response_time_variance
  - msg_waiting_time
  - msg_residence_time
  - …

References

1. Trubiani, Catia, et al. "Automated detection of software performance antipatterns in Java-based applications." *IEEE Transactions on Software Engineering* (2023).
2. VanDonge, Riley, and Naser Ezzati-Jivan. "N-Lane Bridge Performance Antipattern Analysis Using System-Level Execution Tracing." *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022.
3. Chalawadi, Ram Kishan. "Automating the characterization and detection of software performance antipatterns using a data-driven approach." (2021).
4. Stefanakos, Ioannis. *Software analysis and refactoring using probabilistic modelling and performance antipatterns*. Diss. University of York, 2021.
5. Smith, Connie U. "Software performance antipatterns in cyber-physical systems." *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020.
6. Avritzer, Alberto, et al. "Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests." *Journal of Systems and Software* 165 (2020): 110564.
7. Trubiani, Catia, et al. "Exploiting load testing and profiling for performance antipattern detection." *Information and Software Technology* 95 (2018): 329-345.
8. Heger, Christoph, et al. "Expert-guided automatic diagnosis of performance problems in enterprise applications." *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016.
9. Keck, Philipp, et al. "Antipattern-based problem injection for assessing performance and reliability evaluation techniques." *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016.
10. Wert, Alexander, et al. "Automatic detection of performance anti-patterns in inter-component communications." *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*. 2014.

Thanks for your attention

Masoumeh.nourollahi@polymtl.ca
https://github.com/mnourollahi

Blob Injection

- Instead of fetching only the necessary data (such as post IDs) from MongoDB, it fetches entire documents for the user's timeline. This approach unnecessarily transfers large amounts of data between the database and the application, leading to inefficient resource utilization and slower performance.

```
bool found = mongoc_cursor_next(cursor, &doc);
if (found) {
  bson_iter_t iter_0;
  bson_iter_t iter_1;
  bson_iter_t post_id_child;

  bson_iter_t timestamp_child;
  int idx = 0;
  bson_iter_init(&iter_0, doc);

  bson_iter_init(&iter_1, doc);
  while (bson_iter_find_descendant(&iter_0,
          ("posts." + std::to_string(idx) + ".post_id").c_str(),
          &post_id_child)
      && BSON_ITER_HOLDS_INT64(&post_id_child)
      && bson_iter_find_descendant(&iter_1,
          ("posts." + std::to_string(idx) + ".timestamp").c_str(),
          &timestamp_child)
      && BSON_ITER_HOLDS_INT64(&timestamp_child)) {
    auto curr_post_id = bson_iter_int64(&post_id_child);
    auto curr_timestamp = bson_iter_int64(&timestamp_child);
    if (idx >= mongo_start) {
      post_ids.emplace_back(curr_post_id);


    }
    redis_update_map.insert(std::make_pair(std::to_string(curr_timestamp),
        std::to_string(curr_post_id)));
    bson_iter_init(&iter_0, doc);
    bson_iter_init(&iter_1, doc);
    idx++;
  }
}
```

```
bool found = mongoc_cursor_next(cursor, &doc);
if (found) {
  bson_iter_t iter;

  if (bson_iter_init(&iter, doc)) {

    bson_iter_t array;

    if (bson_iter_find_descendant(&iter, "posts", &array)) {

      bson_iter_t post;
      while (bson_iter_next(&array)) {
        bson_iter_recurse(&array, &post);
        bson_iter_t post_id_iter;

        if (bson_iter_find_descendant(&post, "post_id", &post_id_iter)) {



          int64_t post_id = bson_iter_as_int64(&post_id_iter);


          post_ids.push_back(post_id);
        }
      }
    }


  }
}
```

Empty-semi-truck Injection

- After retrieving data from Redis, a large vector containing 100 MB of unnecessary data is created
- By introducing this anti-pattern, we illustrate how transferring large amounts of unnecessary data can lead to increased resource consumption, slower performance, and wasted resources.

```
// Simulate empty semi-truck anti-pattern: transferring large amounts of data
unnecessarily
std::vector<char> unnecessary_data(1024 * 1024 * 100, '0'); // 100 MB of
unnecessary data
LOG(info) << "Transferring unnecessary data of size: " << unnecessary_data.size()
<< " bytes";

// Transfer the unnecessary data over the network
std::ofstream outfile("unnecessary_data.txt", std::ios::binary);
outfile.write(unnecessary_data.data(), unnecessary_data.size());
outfile.close();

//empty semi-truck finished
```

- After creating the data, it is then written to a file named "unnecessary_data.txt" in binary mode. This file write operation simulates the transfer of the unnecessary data over the network, which impacts the network resources.