P^3: A Profiler Suite for Parallel Applications on the Java Virtual Machine

Andrea Rosà, Walter Binder

Università della Svizzera italiana, Lugano, Switzerland

APLAS 2020

SPLASH 2022
COVID Time Papers In Person
December 6, 2022

Background

- Concurrency is becoming increasingly important to speed up applications
- It is fundamental to analyze **concurrency** and **synchronization** constructs used by concurrent applications
  - Enables performance assessment
  - Enables detection of optimization opportunities

- Novel **profiling suite** for parallel applications running on the Java Virtual Machine (JVM)
- Focus on metrics related to parallelism, concurrency, and synchronization
  - Concurrent entities (e.g., threads, tasks, actors, futures)
  - Constructs and classes to implement synchronization (e.g., locks, parking, synchronizers)
  - Lock-free operations (e.g., atomic, volatile)
  - Synchronized and concurrent collections

- P3 can be readily applied:
  - To popular benchmark suites
  - To public code repositories
- P3 incurs only moderate profiling overhead

- Challenges in developing P3:
  - Moderate overhead
  - High accuracy
- Enabling features:
  - Use of lock-free data structures
  - Few computations done in instrumentation
  - Use of reification of reflective information in a separate instrumentation process [1]

[1] A. Rosà et al., "Optimizing Type-specific Instrumentation on the JVM with Reflective Supertype Information". Journal of Visual Languages & Computing 49, 2018.

Outline

1. Metrics
2. Architecture
3. Implementation
4. Application to previous research
5. Evaluation
6. Conclusions

Metrics

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics – Concurrent Entities

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics - Synchronization

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

Metrics – Lock-free Operations

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

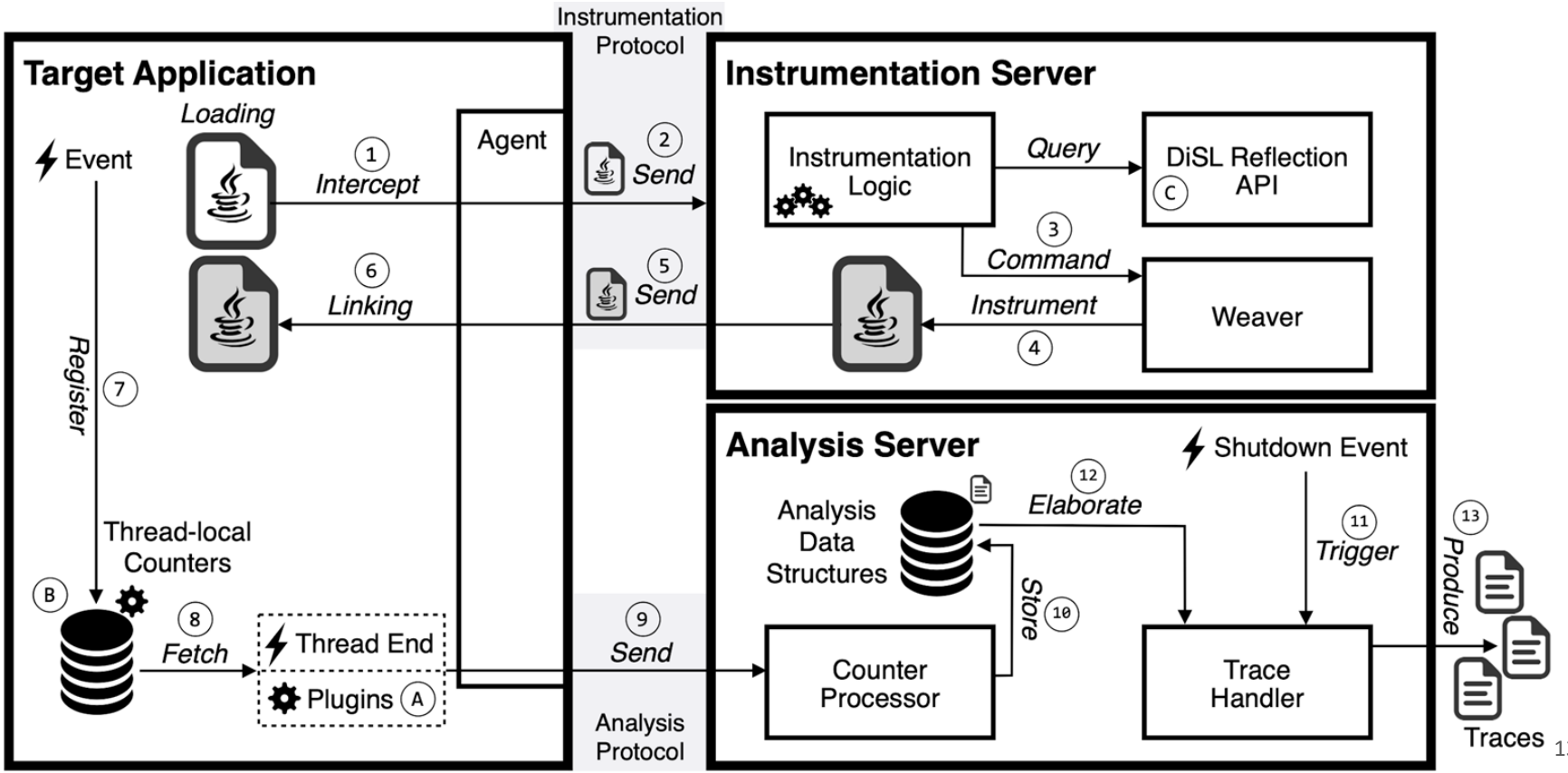
Metrics - Collections

Module	Metrics
thread	Threads start and termination.
task	Tasks creation and execution (instances of Runnable, Callable and ForkJoinTask).
actor	Use of Akka actors.
future	Futures and promises from Java's, Scala's and Twitter's libraries.
ilock	Implicit locks: use of synchronized methods and blocks.
elock	Explicit locks: use of interfaces Lock, ReadWriteLock and Condition.
wait	Calls to Object.wait, Object.notify and Object.notifyAll.
join	Calls to Thread.join.
park	Thread parking and unparking.
synch	Synchronizers: Semaphore, CountDownLatch, CyclicBarrier, Phaser and Exchanger.
cas	Compare-and-swap (CAS), get-and-swap (GAS), get-and-add (GAA).
atomic	Use of atomic classes: AtomicInt, AtomicLong, AtomicReference.
volatile	Accesses to volatile fields.
scoll	Use of synchronized collections.
ccoll	Use of concurrent collections: BlockingQueue, ConcurrentMap and subtypes.

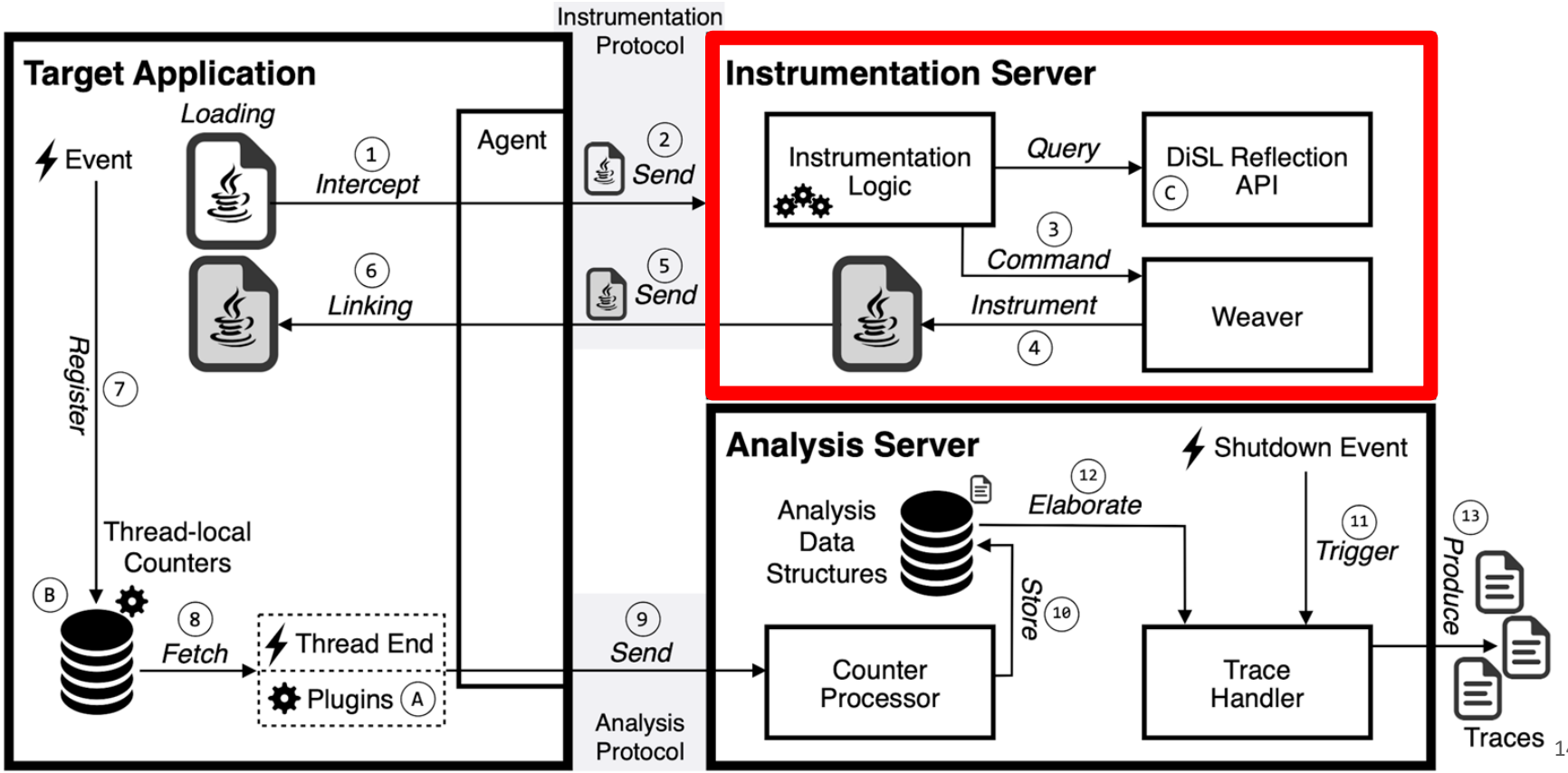
Additional Metrics

- Bytecode count
  - Number of bytecode instructions executed
  - Allows metric normalization w.r.t. platform-independent quantity
  - Useful for comparing metrics in different applications
- Caller context
  - Method in which an event occurs
  - Allows per-method event counters
  - Enable detection of code where most events of a given type occur
    - Useful information to locate optimization opportunities

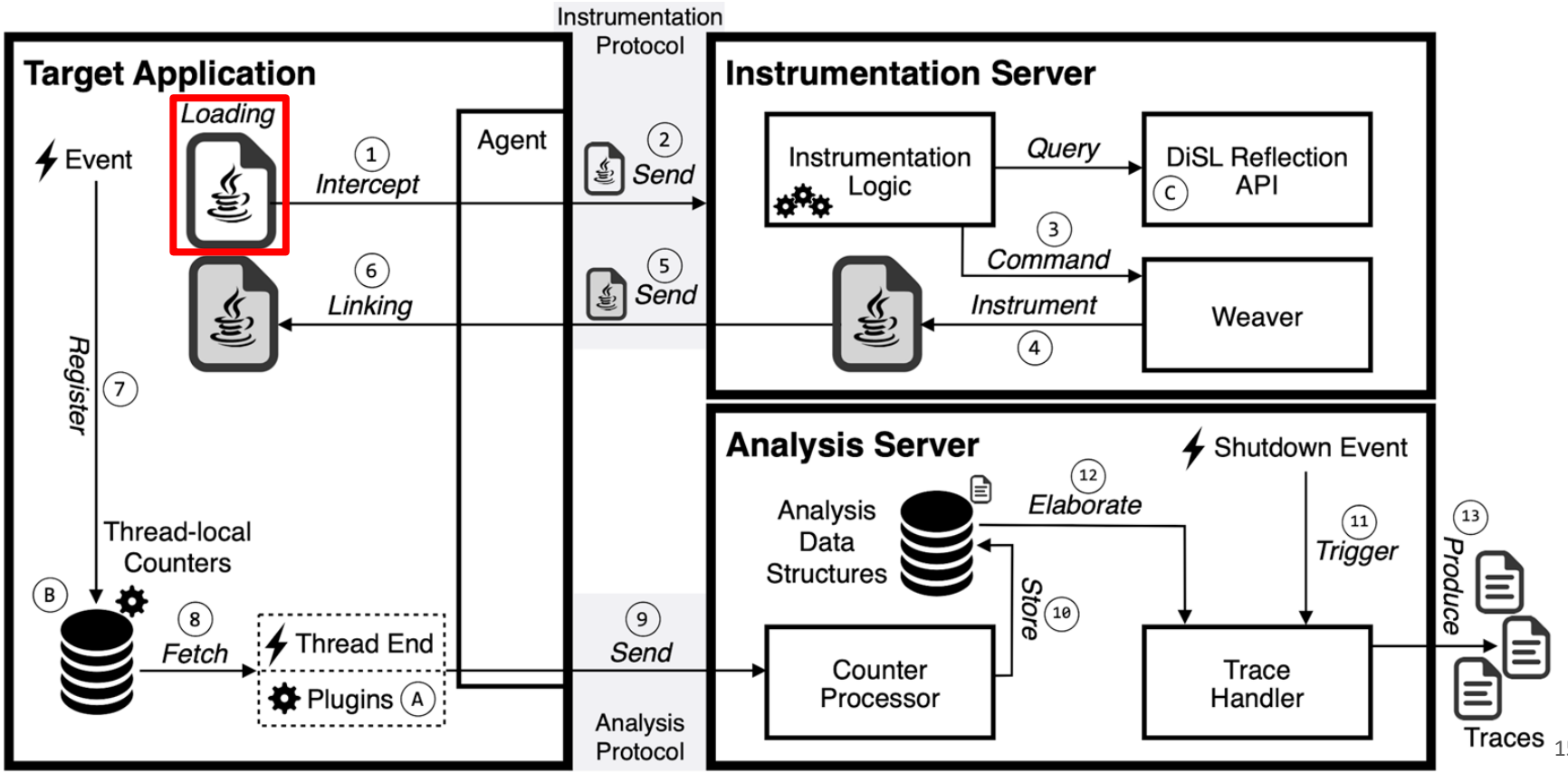
Architecture



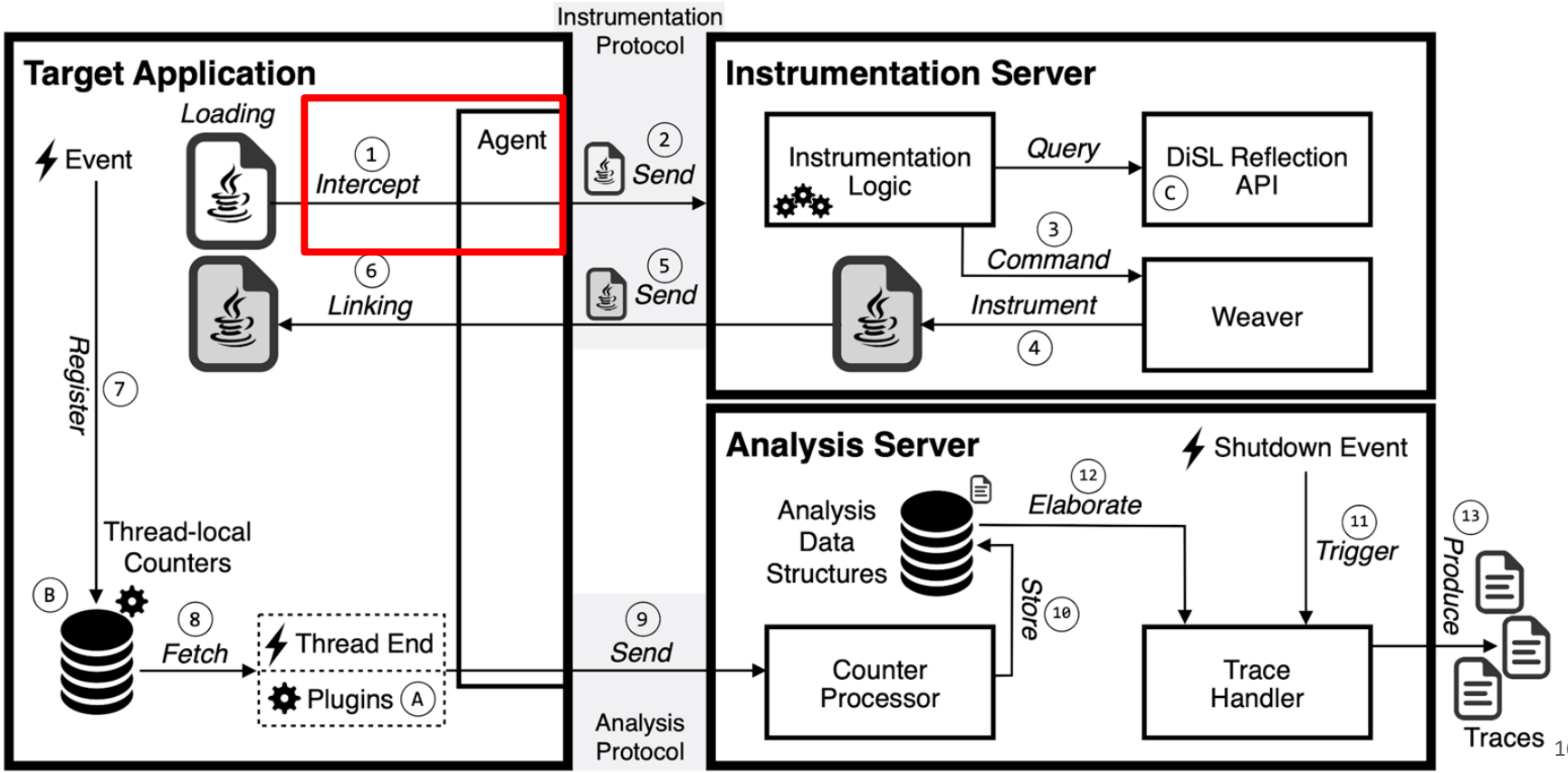
Architecture



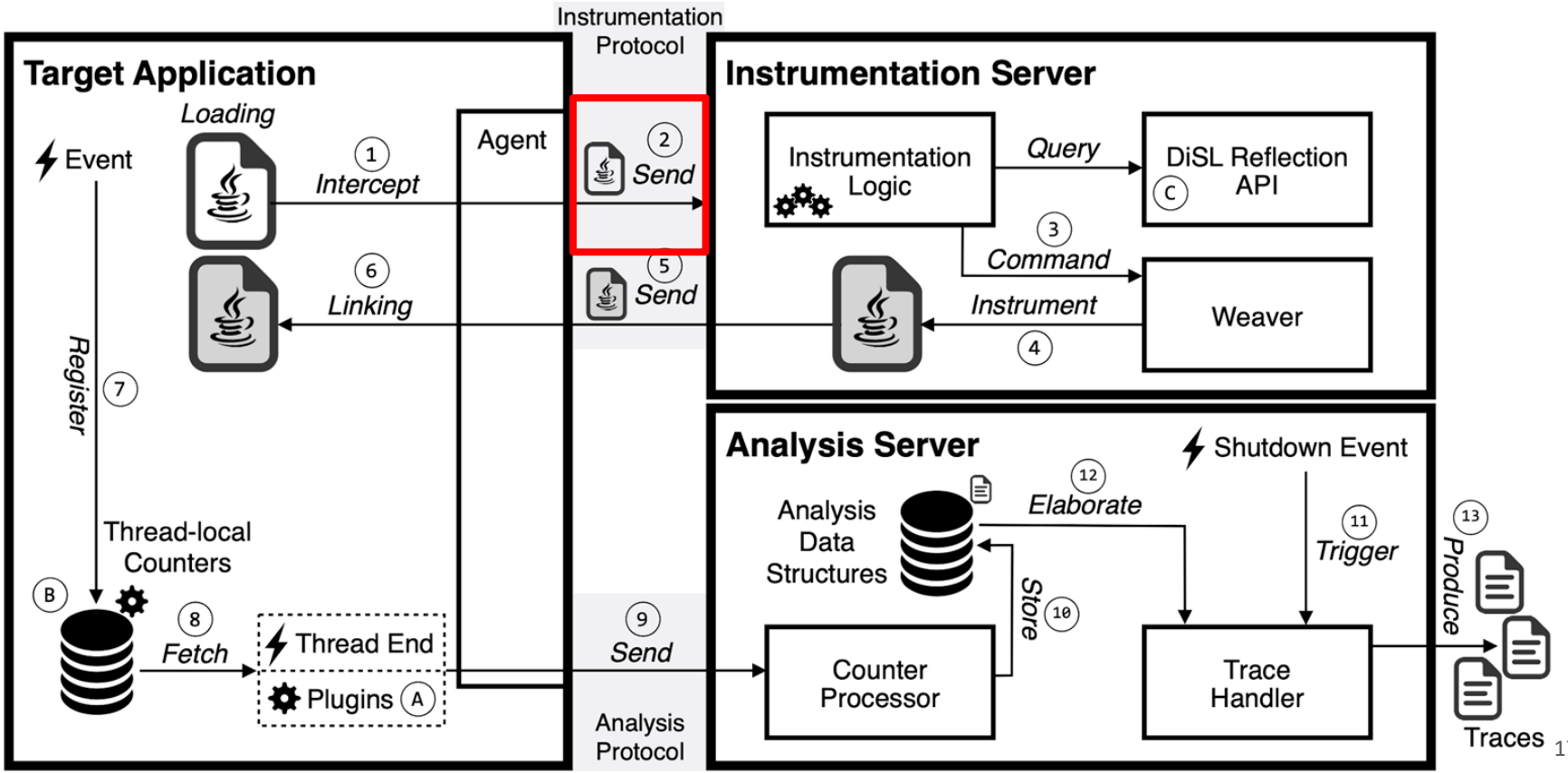
Architecture



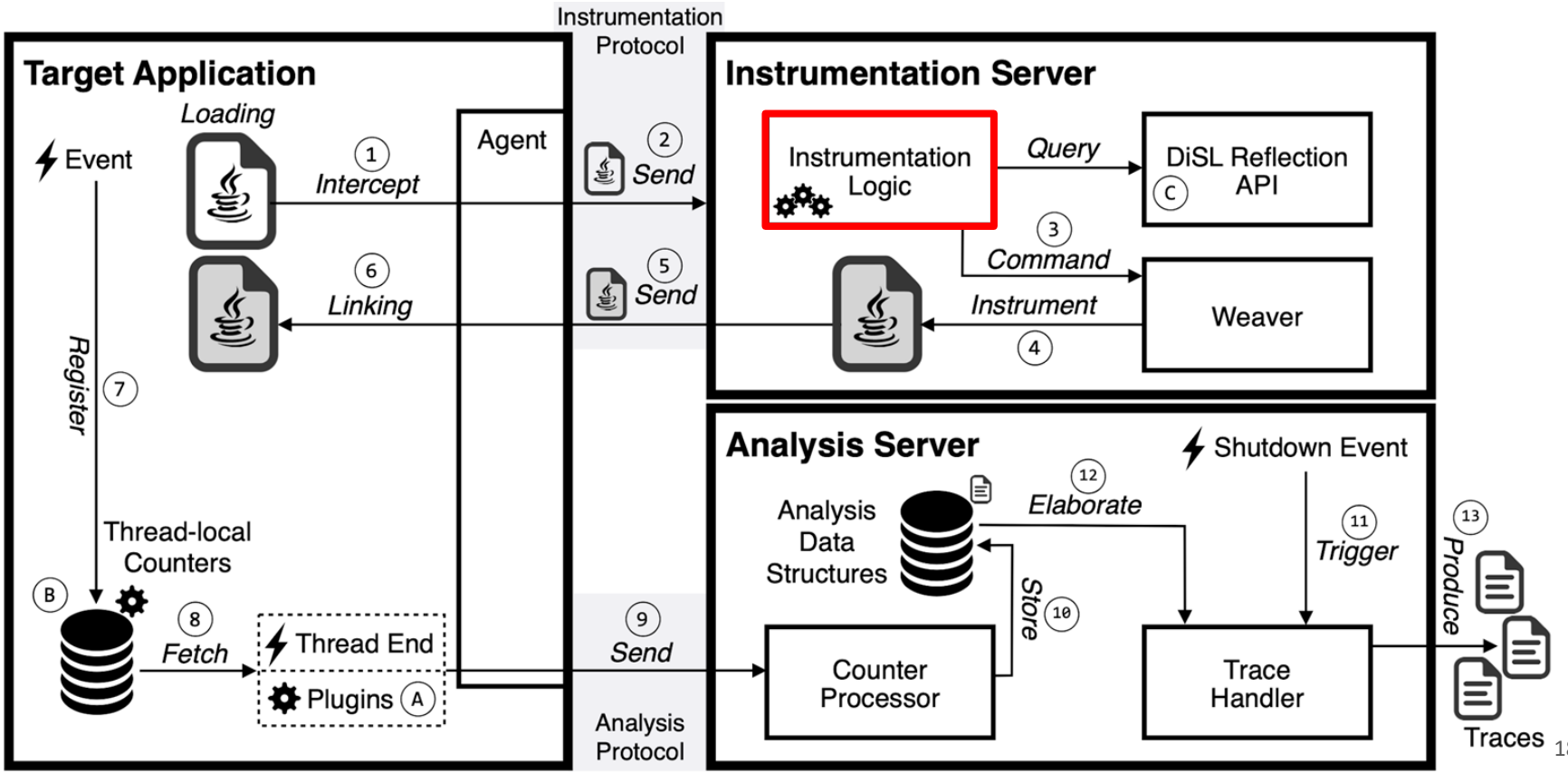
Architecture



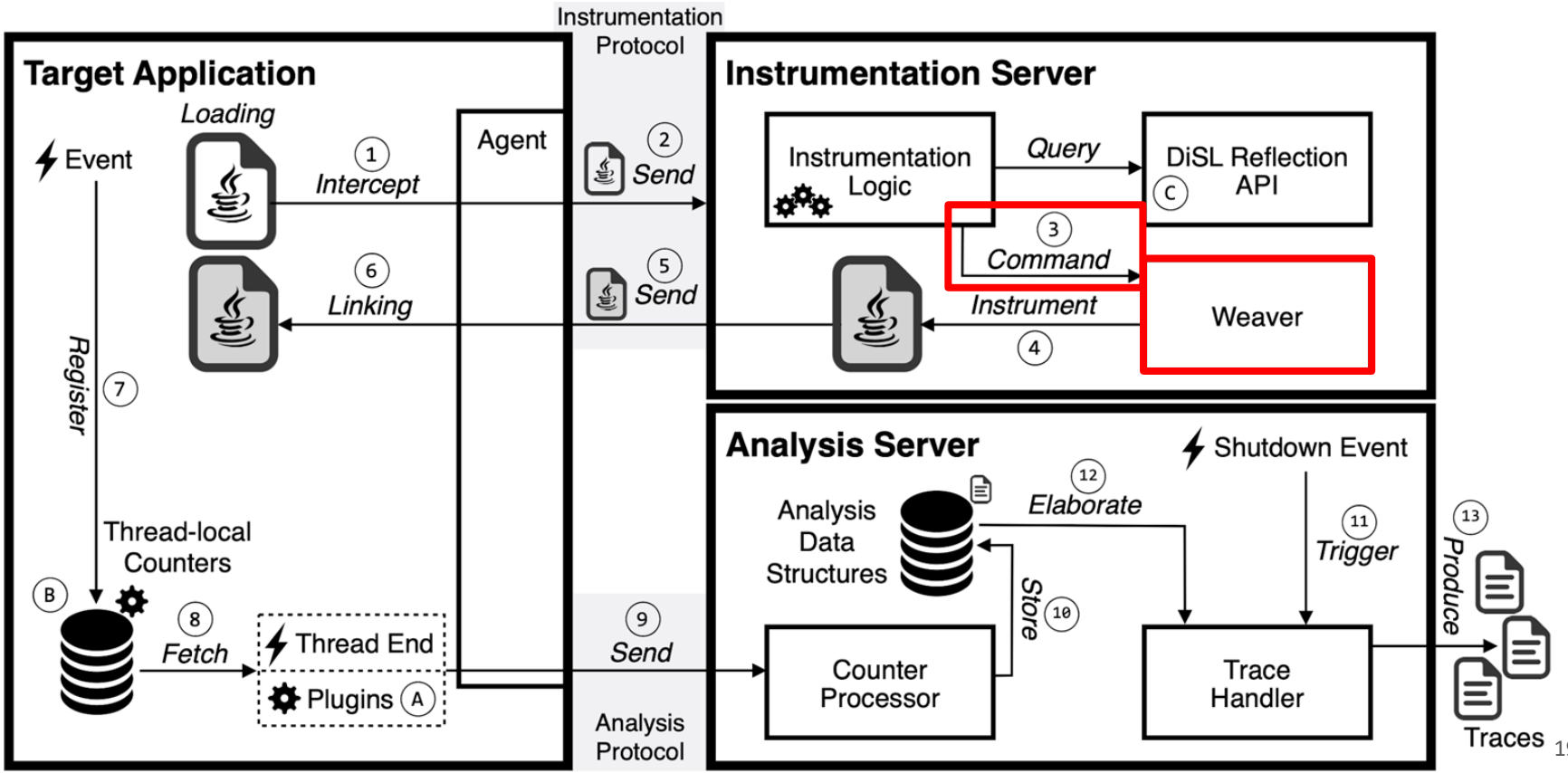
Architecture



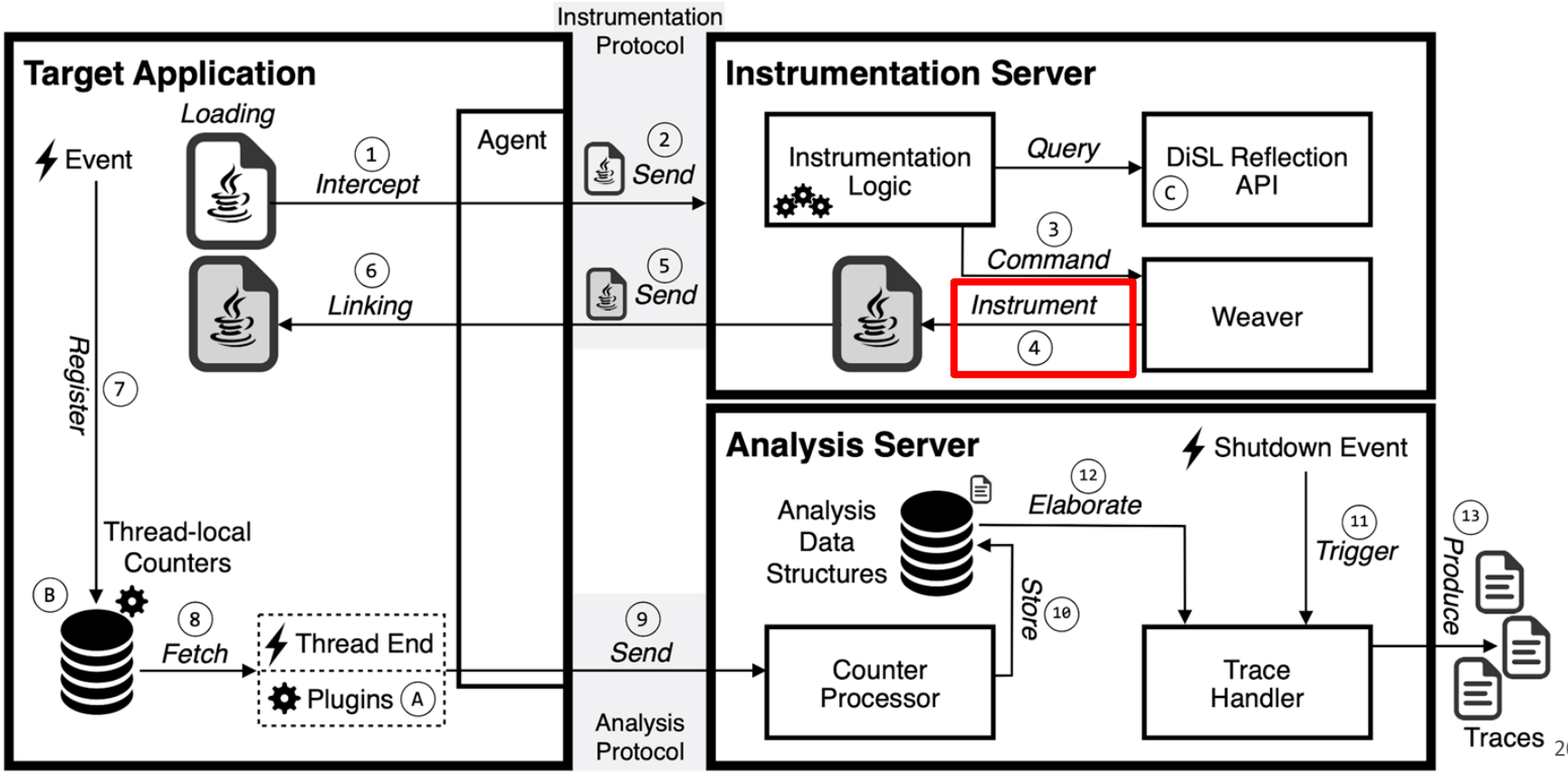
Architecture



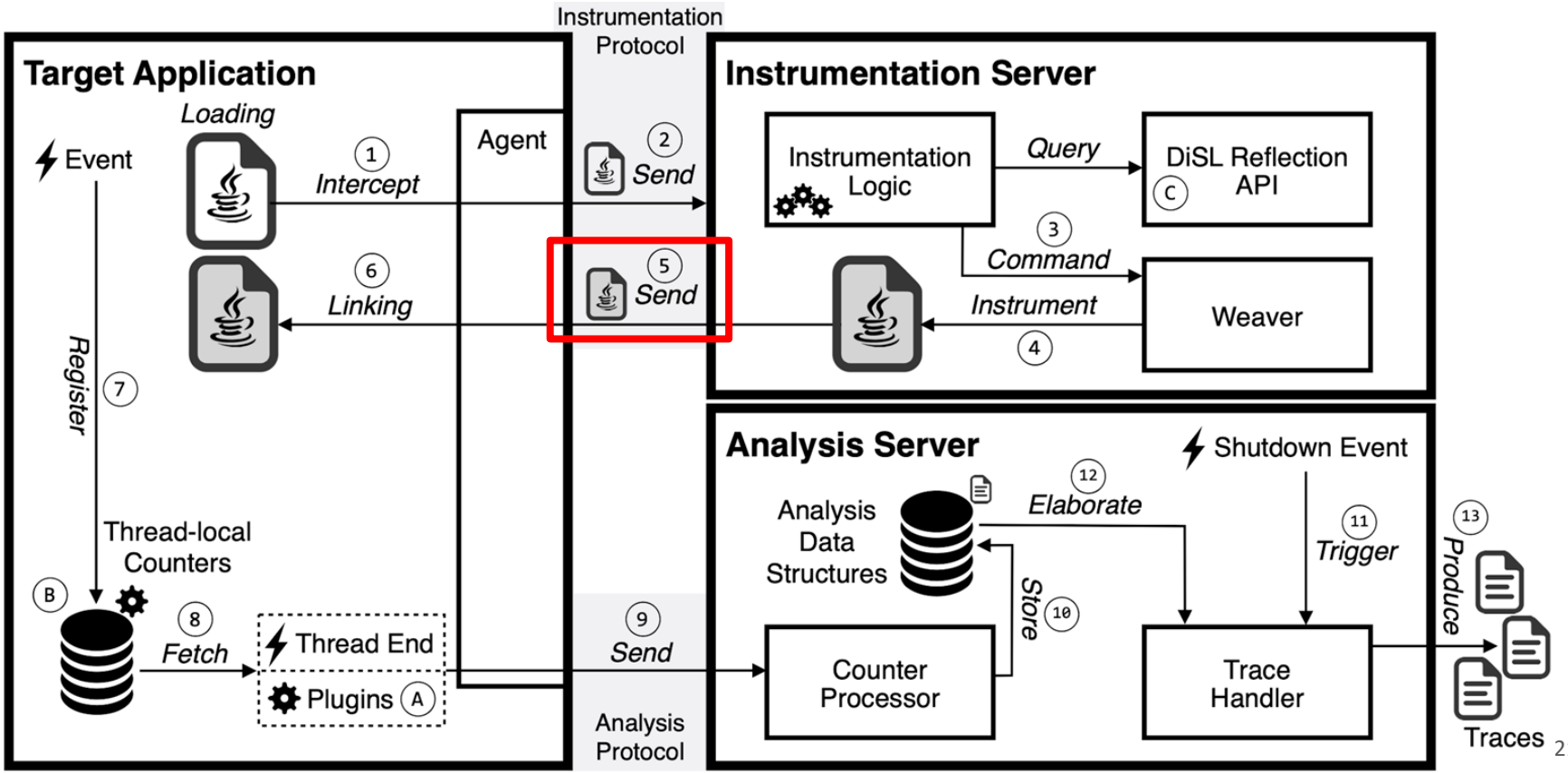
Architecture



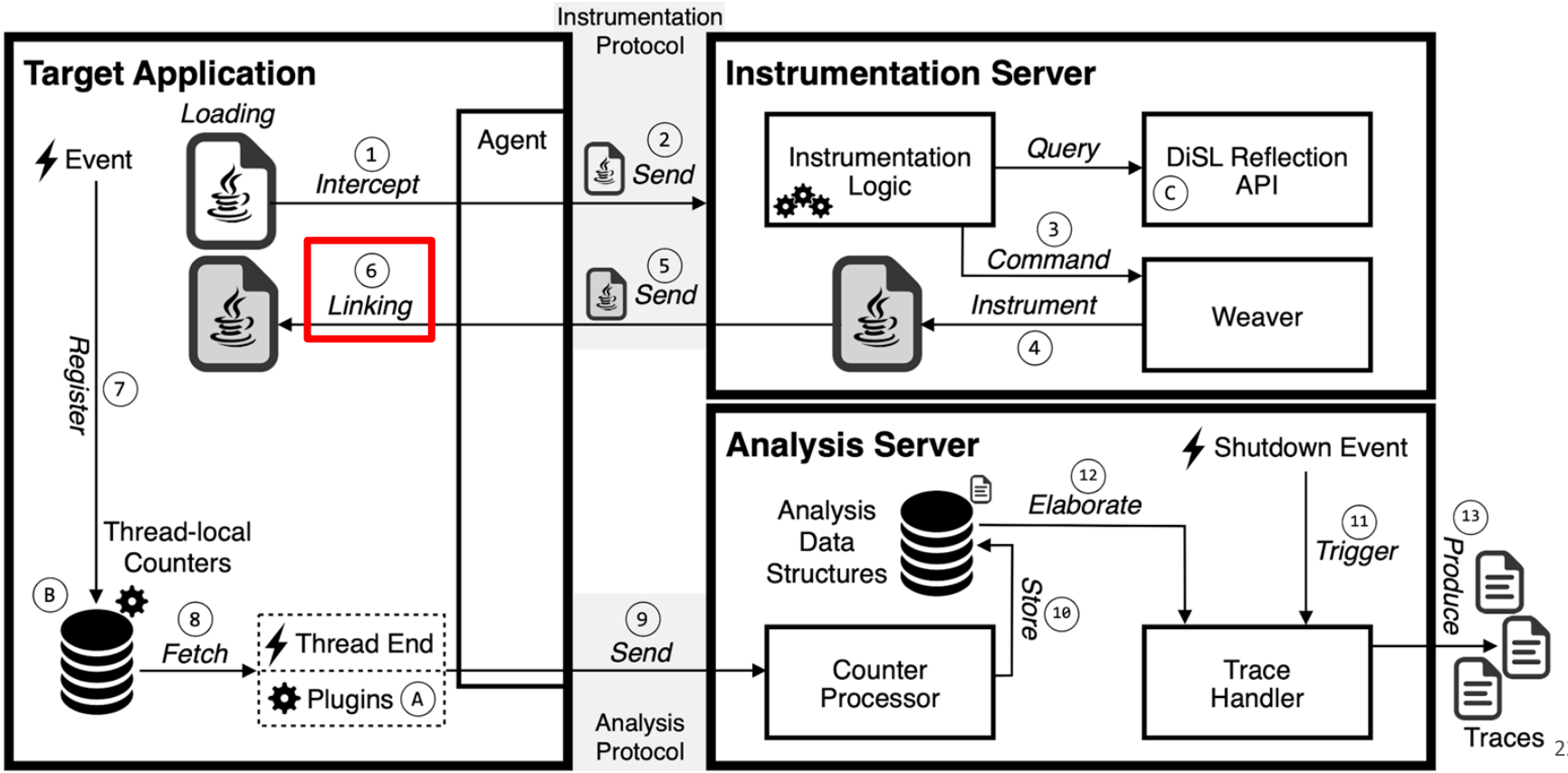
Architecture



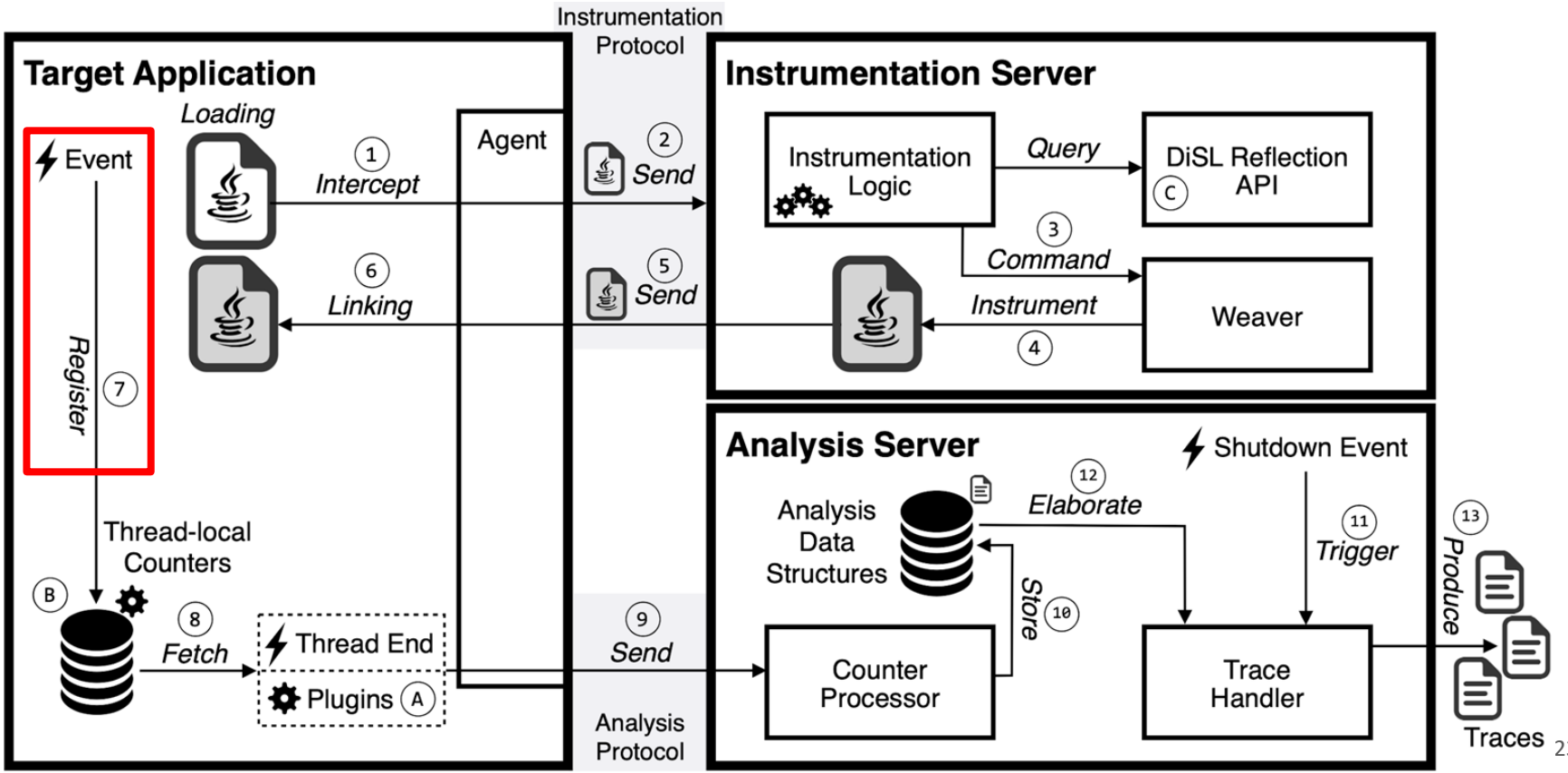
Architecture



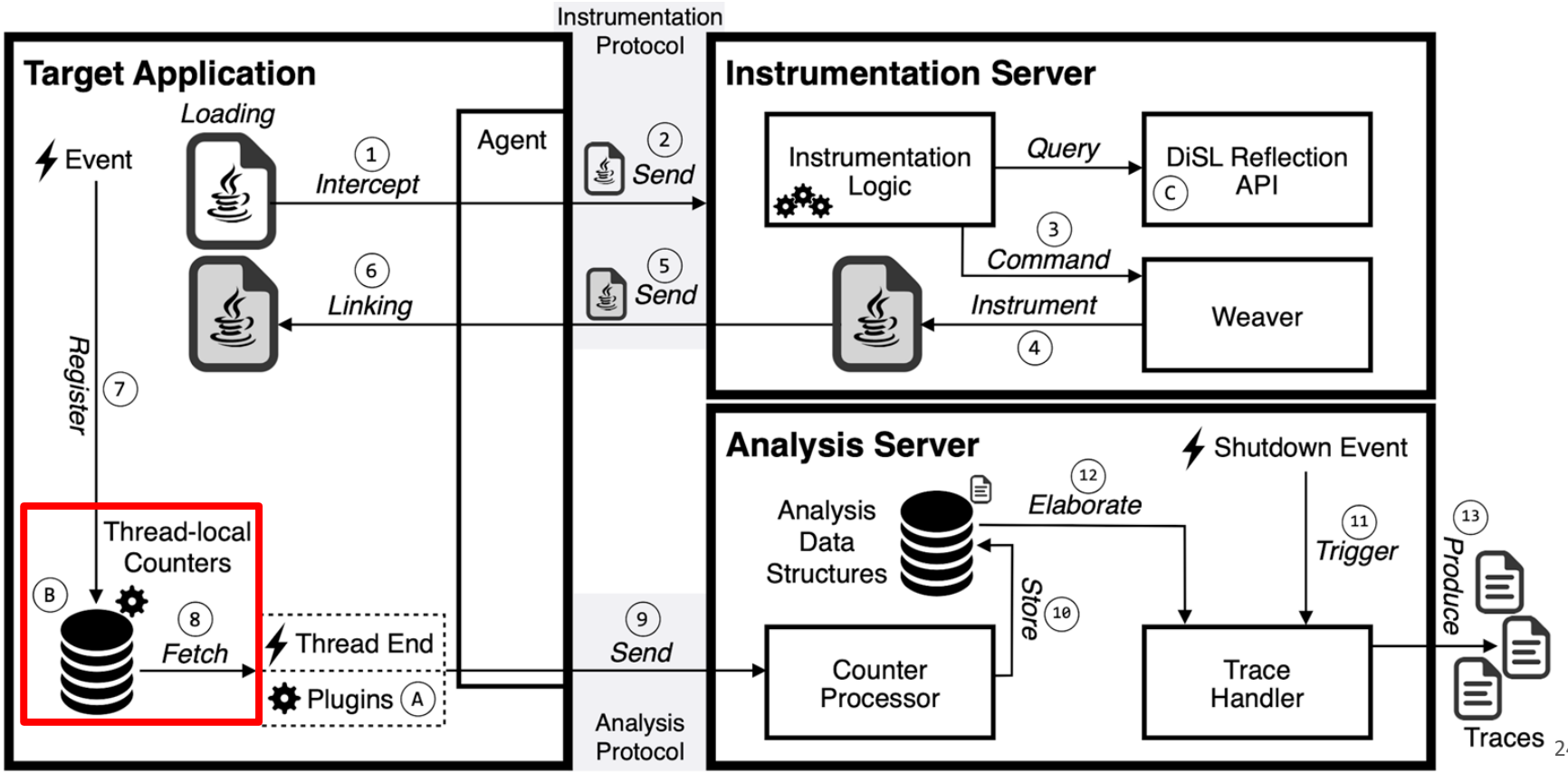
`Architecture



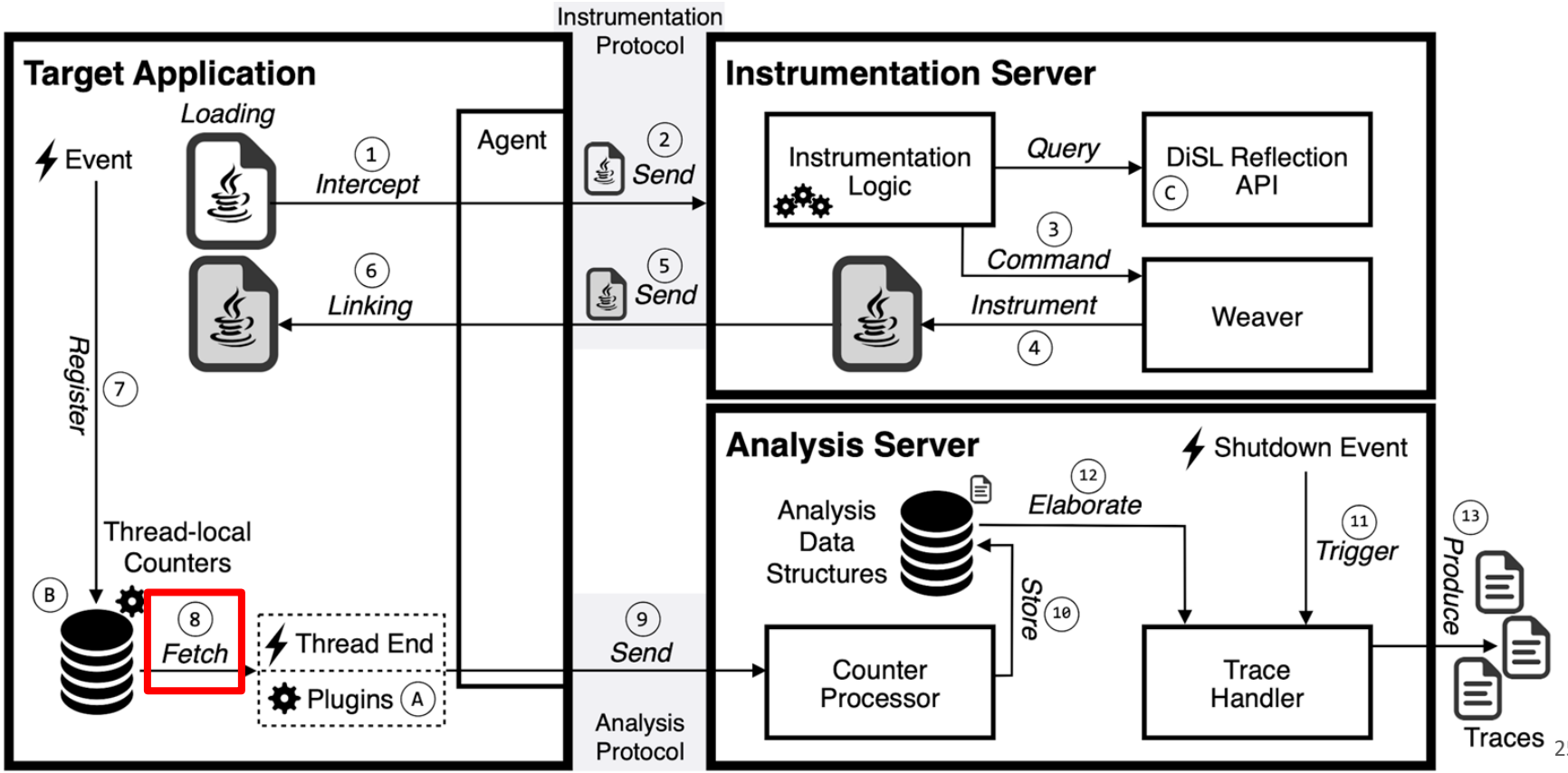
Architecture



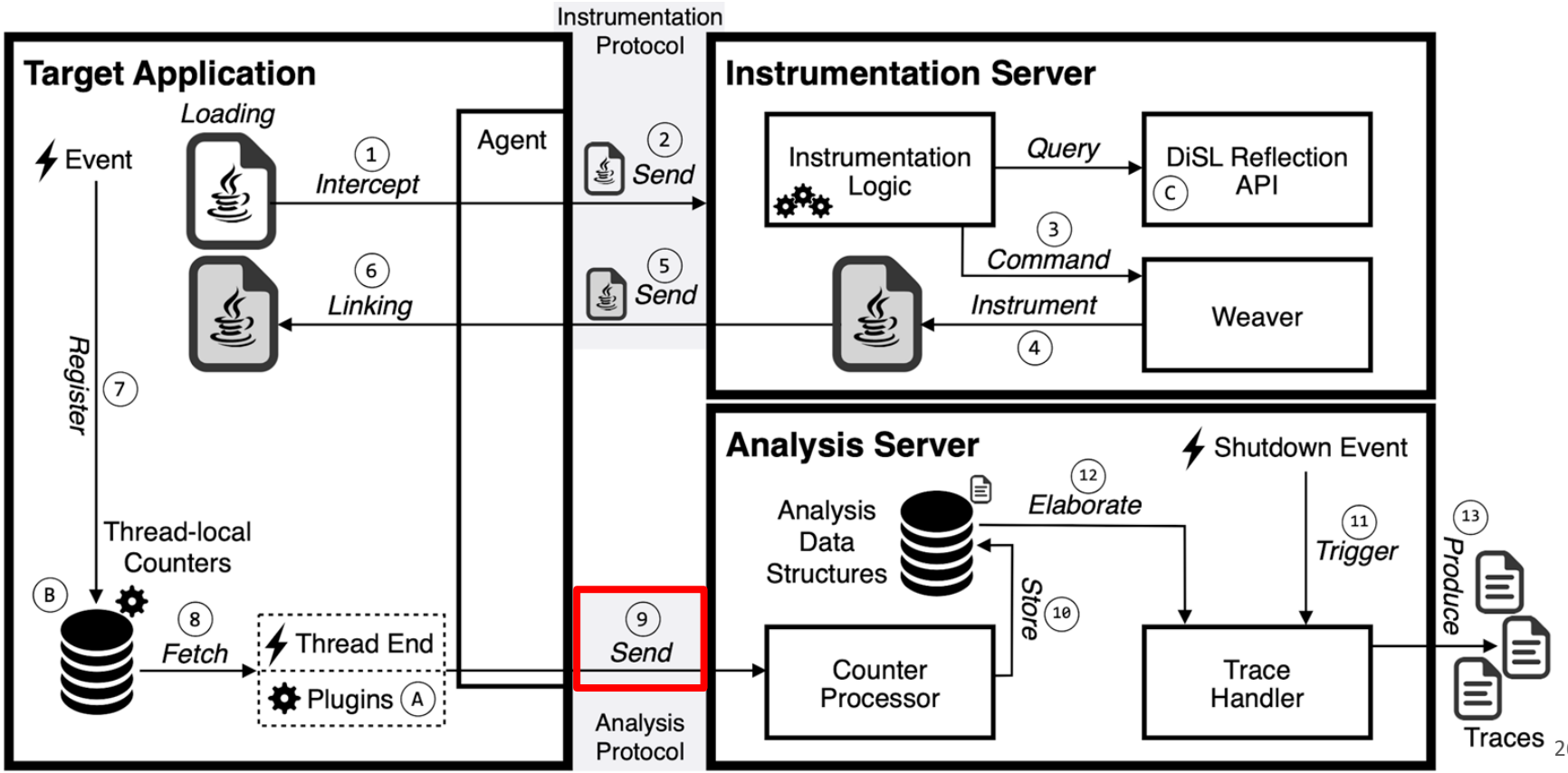
Architecture



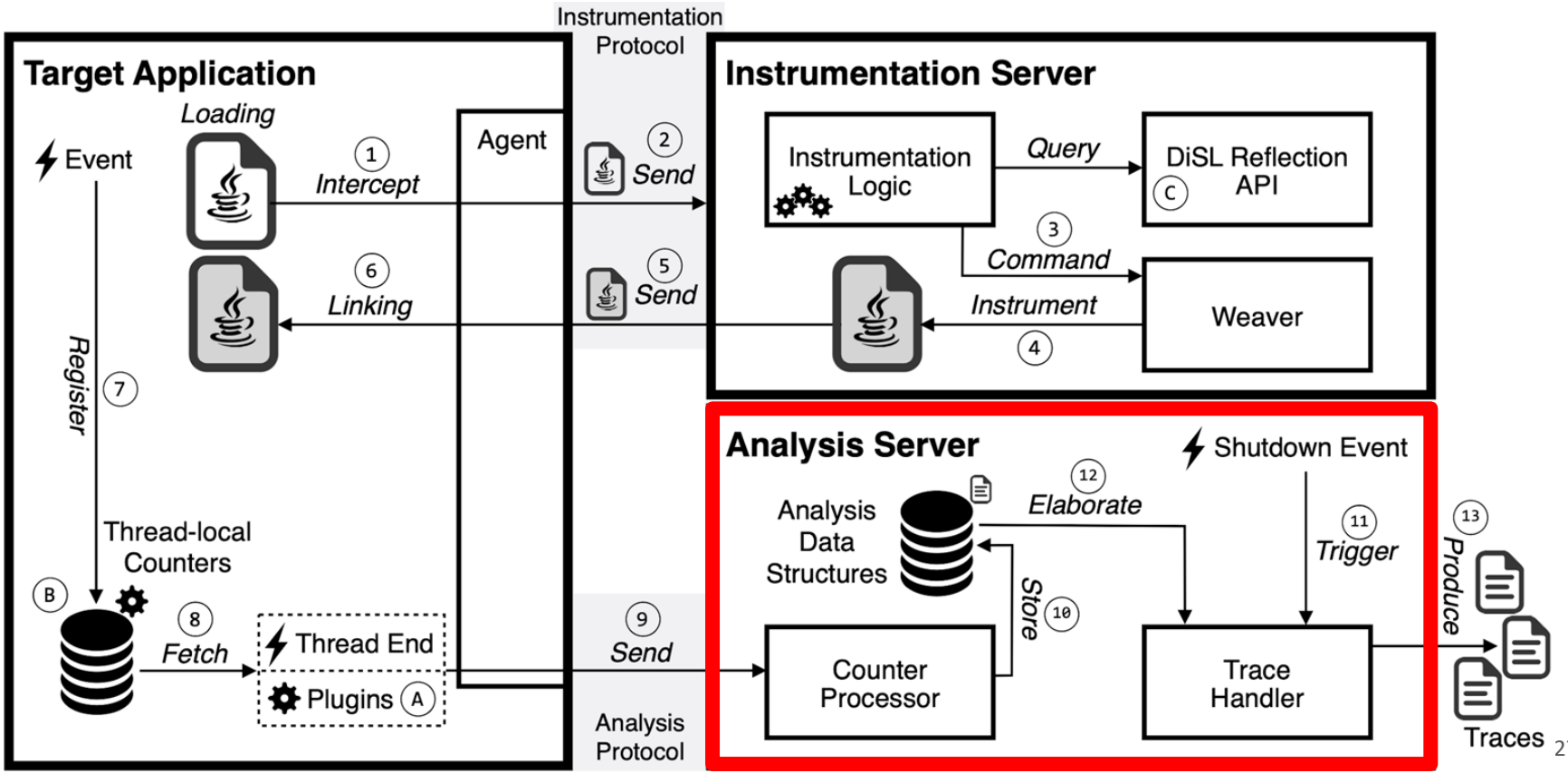
Architecture



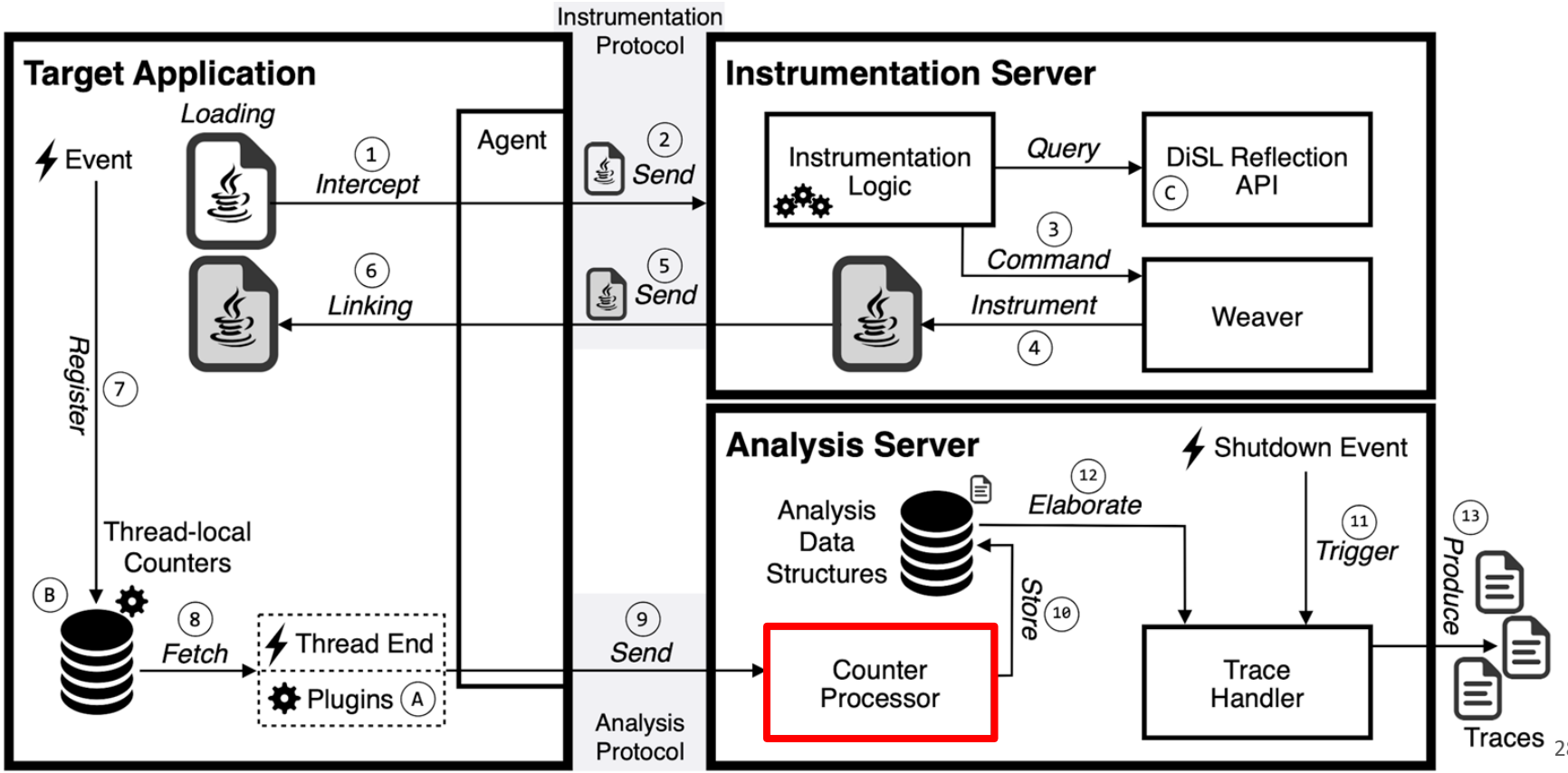
Architecture



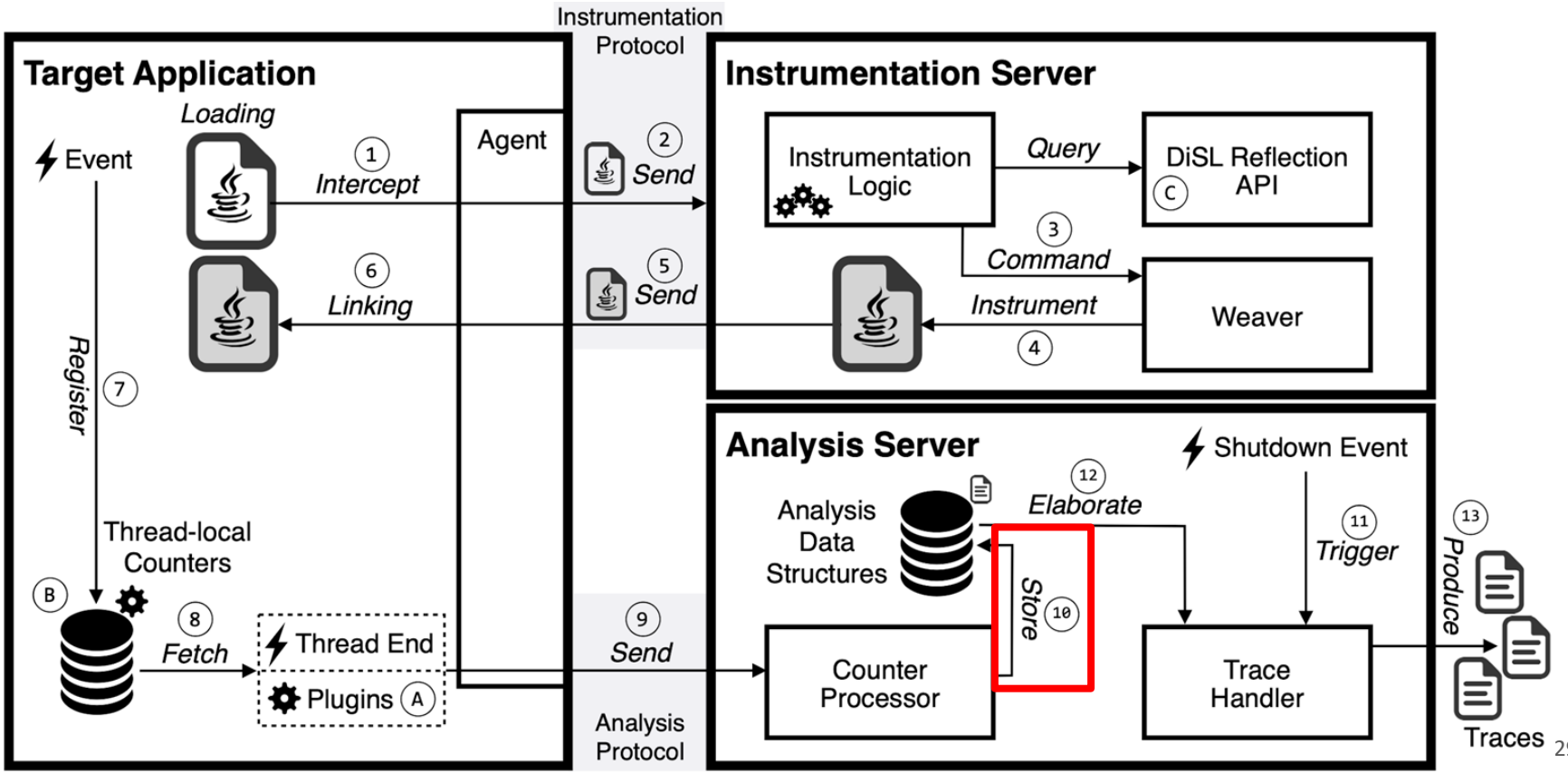
Architecture



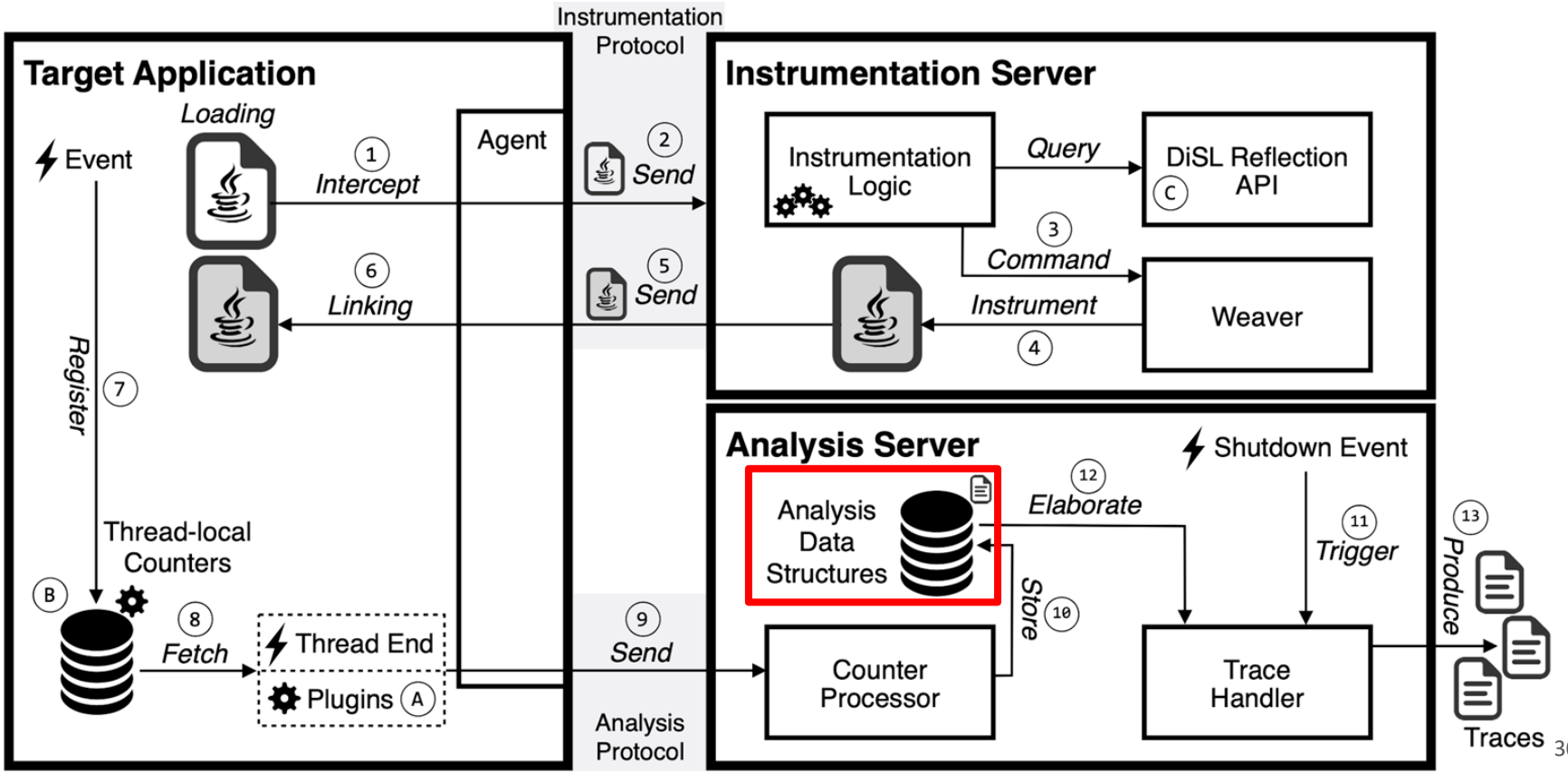
Architecture



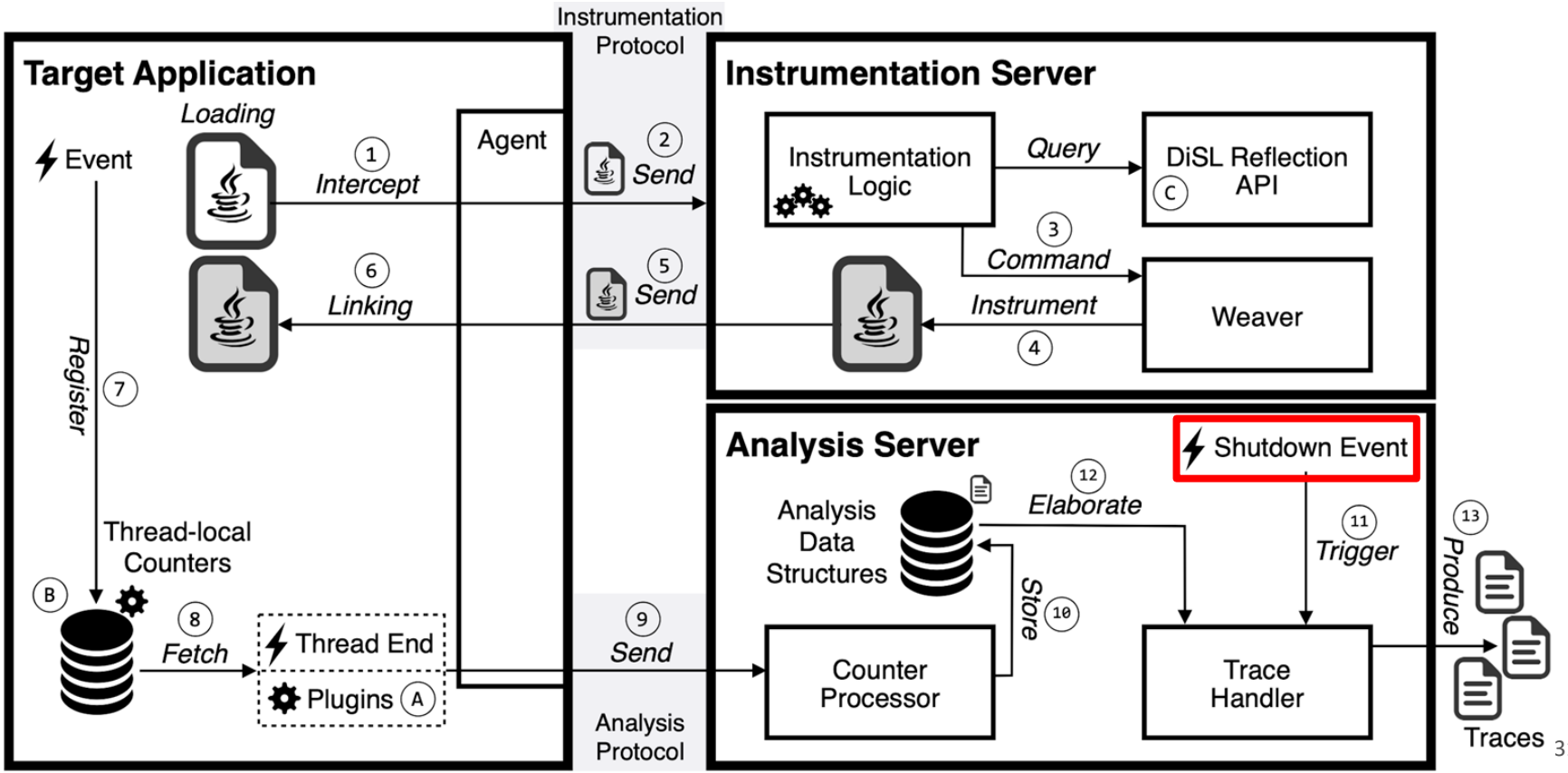
Architecture



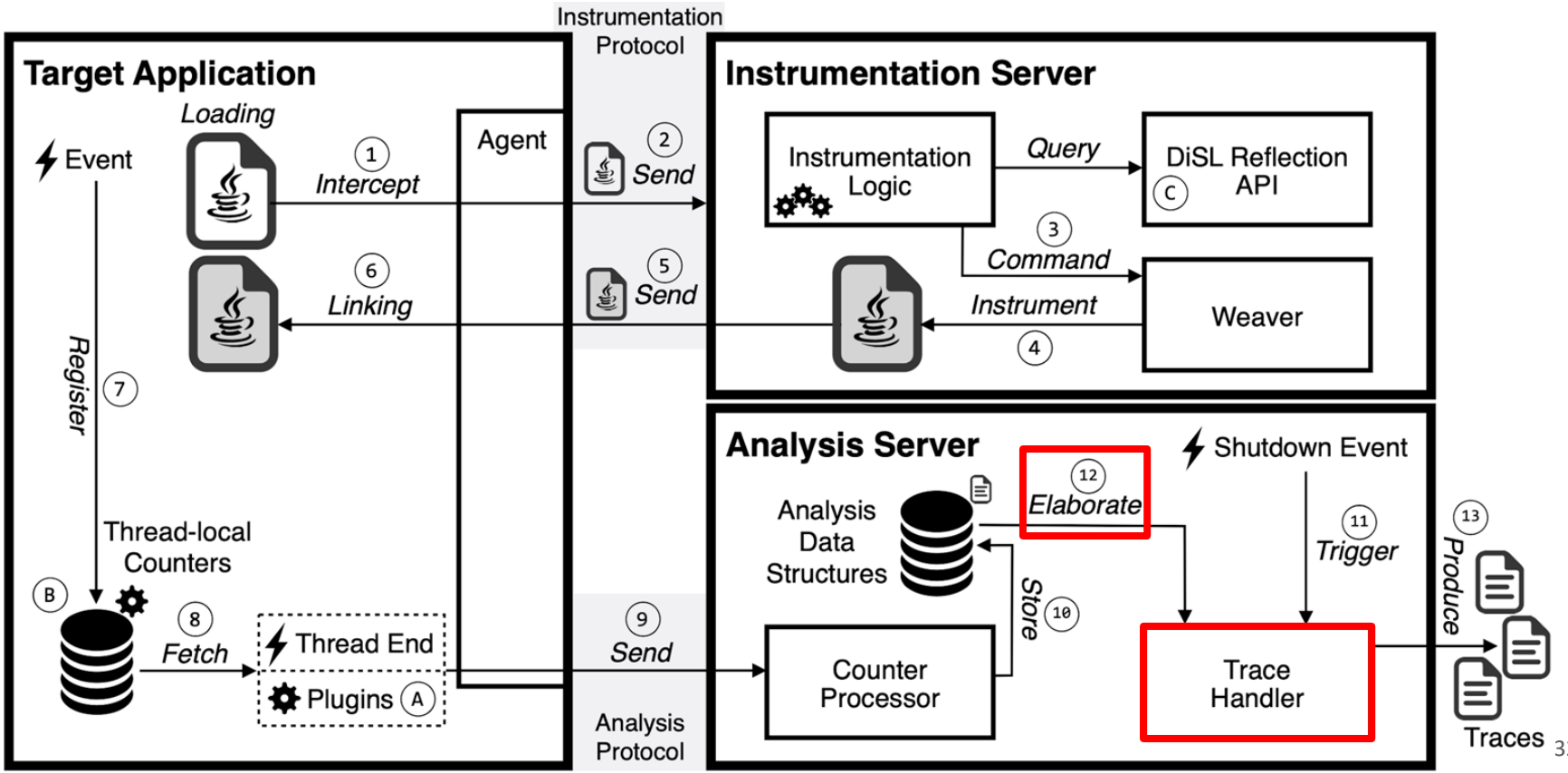
Architecture



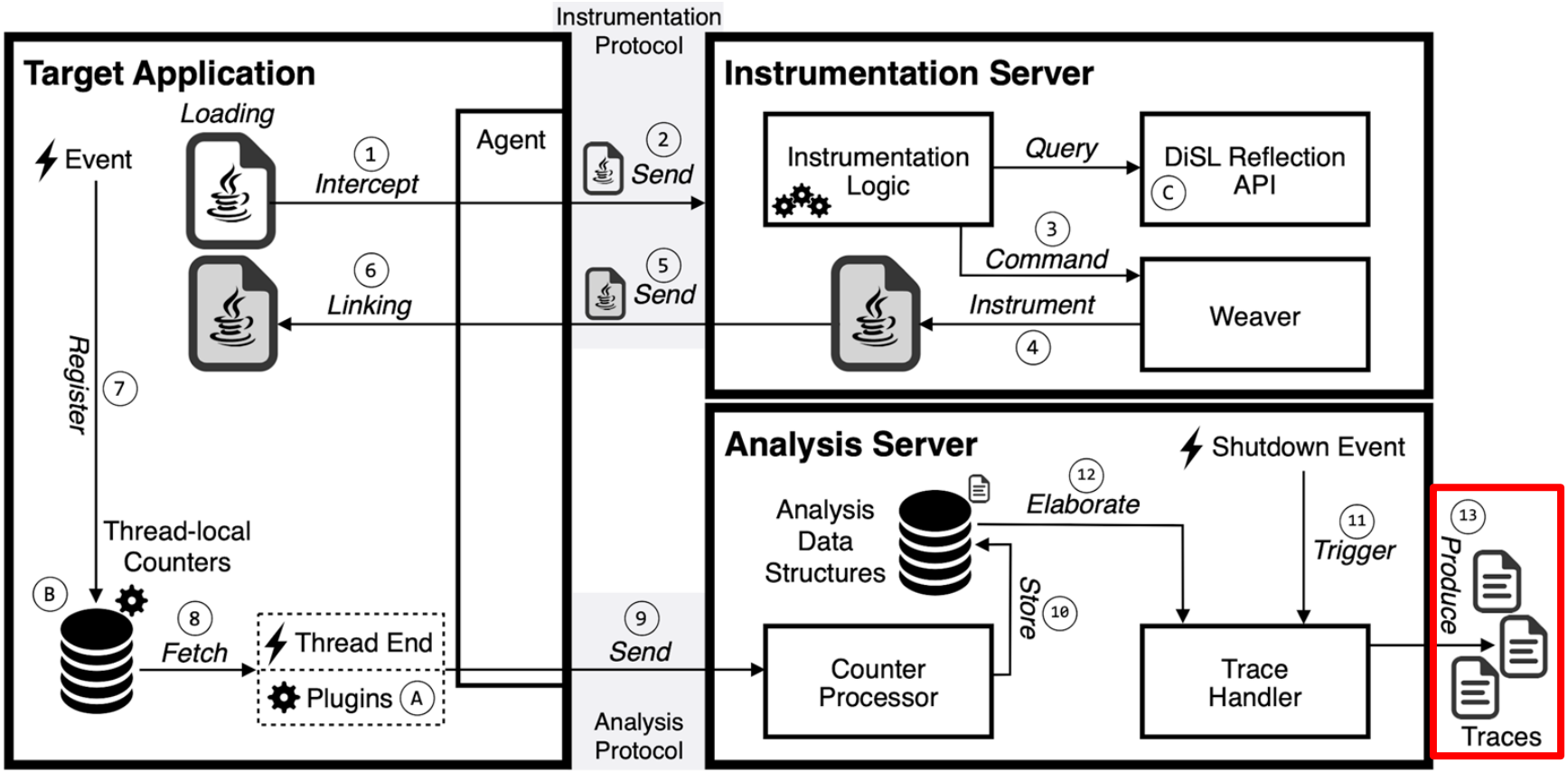
Architecture



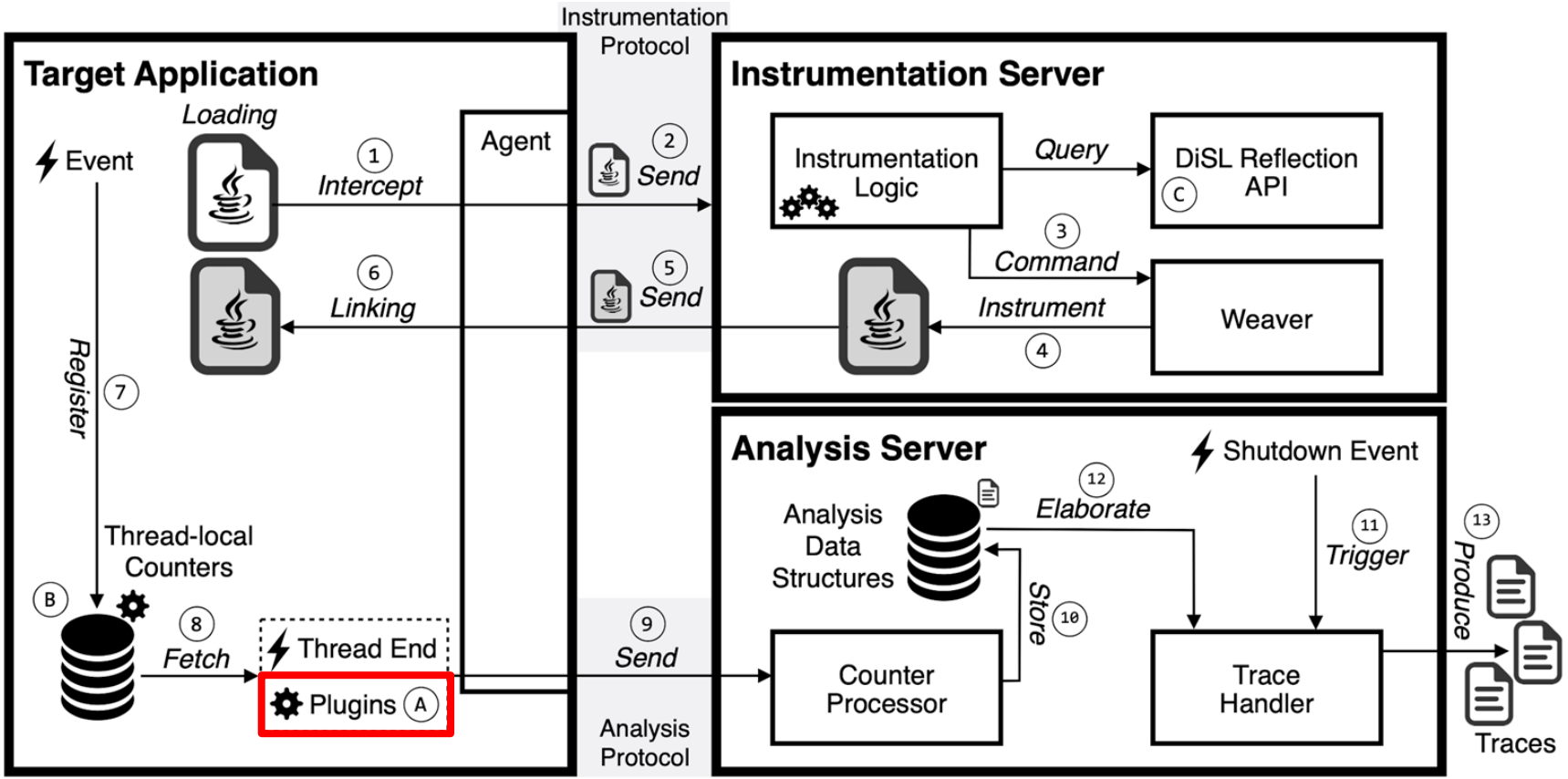
Architecture



Architecture



Architecture



Plugins

- Allow determining benchmark iteration start/end
  - Enable collection of per-iteration metrics
  - Useful for differentiating warm-up from steady-state performance
- P^3 includes plugins for Renaissance [1], DaCapo [2], ScalaBench [3], SPECjvm2008 [4]
- Users can implement plugins for other benchmark suites
- Can interface with NAB [5]
  - A framework for conducting dynamic analysis on public code repositories

[1] A. Prokopec et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019.
[2] S. Blackburn et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". SIGPLAN Not. 41(10), 2006.
[3] A. Sewe et al., "Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine". OOPSLA 2011.
[4] SPECjvm2008. `https://www.spec.org/jvm2008/`
[5] A. Villazon et al., "Automated Large-scale Multi-language Dynamic Program Analysis in the Wild". ECOOP 2019.

Main Implementation Details

- Built on top of the DiSL framework for bytecode instrumentation [1]
  - Guarantees complete bytecode coverage
  - Events of interest detectable also in the Java Class Library
- Implementation designed to keep profiling overhead moderate while not jeopardizing accuracy
  - Events registered in thread-local primitive counters
    - No expensive synchronization or extra heap allocations
  - Counter elaboration done in a separate process

[1] L. Marek et al., " DiSL: A Domain-specific Language for Bytecode Instrumentation". AOSD 2012.

Main Implementation Details

- P^3 requires access to reflective information on class under instrumentation
  - Usually not available in out-of-process instrumentation
  - Usual expensive solutions in instrumentation code:
    - Insertion of expensive dynamic checks
    - Use of Java Reflection API
- P^3 can access reflective information thanks to the DiSL Reflection API [1]
  - Provides partial reflective information on class under instrumentation
  - Greatly reduces profiling overhead
    - E.g., from 1613x to 1.03x (volatile)

[1] A. Rosà et al., "Optimizing Type-specific Instrumentation on the JVM with Reflective Supertype Information". Journal of Visual Languages & Computing 49, 2018.

Applications to Previous Research

- P^3 was fundamental in development of Renaissance [1]
  - Selection of candidate workloads in public software repositories
    - Showing high concurrency and synchronization
  - Filter out workloads with low parallelism and concurrency
  - Profile key metrics on concurrency and synchronization
    - Demonstrate diversity of Renaissance wrt. other suites
- P^3 was used to conduct large-scale analyses with NAB [2]
  - Particularly on task-parallel workloads

[1] A. Prokopec et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019.
[2] A. Villazon et al., "Automated Large-scale Multi-language Dynamic Program Analysis in the Wild". ECOOP 2019.

Evaluation

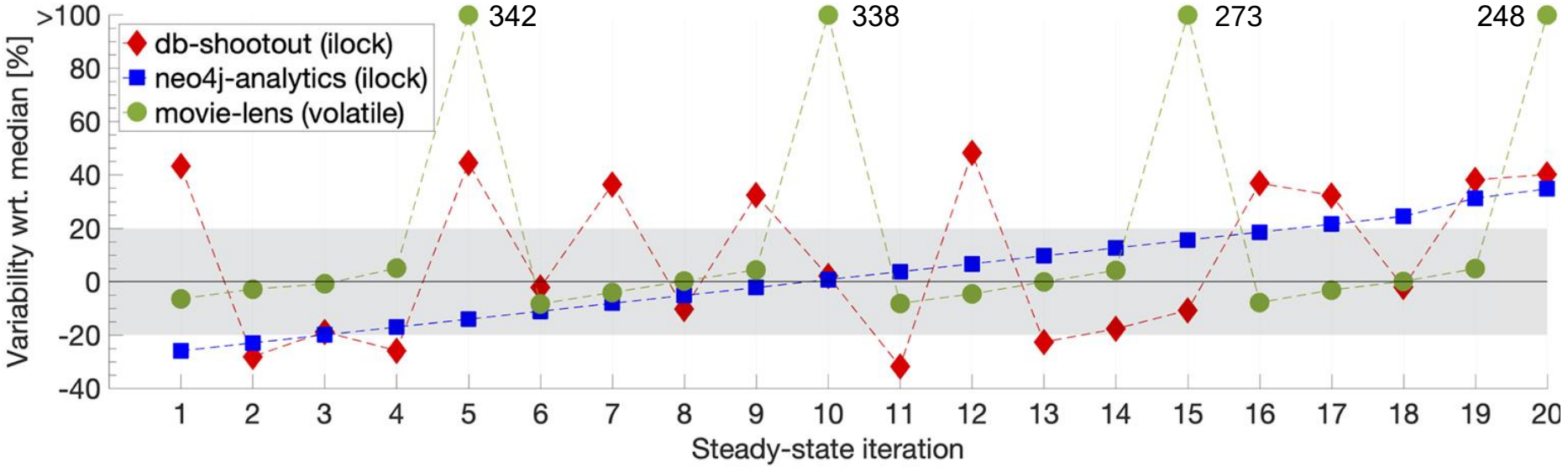
- Target workload: Renaissance benchmark suite [1]
  - Variability of metrics
  - Profiling overhead
- Evaluation setting:
  - Only steady-state iterations considered
  - Instrumentation and analysis servers deployed on different NUMA node than Renaissance
  - No other CPU-, memory-, or IO-intensive applications in execution

[1] A. Prokopec et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019.

Evaluation – Metric Variability

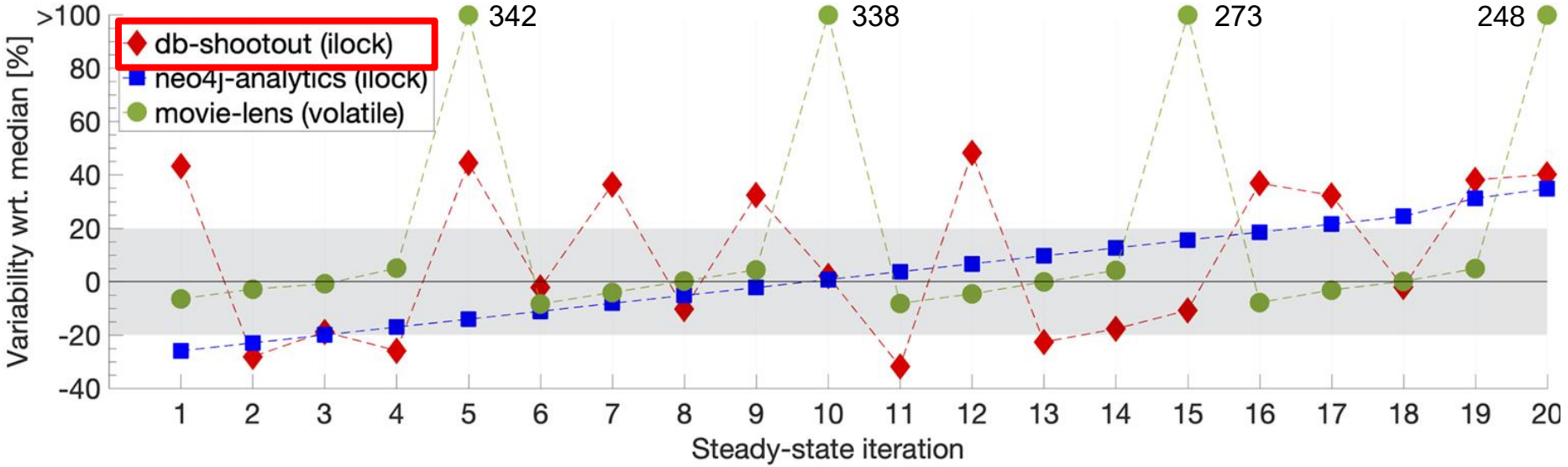
- Goal: Conduct preliminary analysis on metric variability for multiple interations of the Renaissance benchmarks
  - Focus on metrics on parallelism, concurrency and synchronization
  - Aim at finding workloads showing symptoms of metric variability
- Profile all supported metrics in 20 steady-state iterations
- For each metric:
  - Compare values in each iteration with median across all steady-state iterations
  - Focus on benchmarks with a variation >= ±20% wrt. median in at least one iteration

Evaluation – Metric Variability



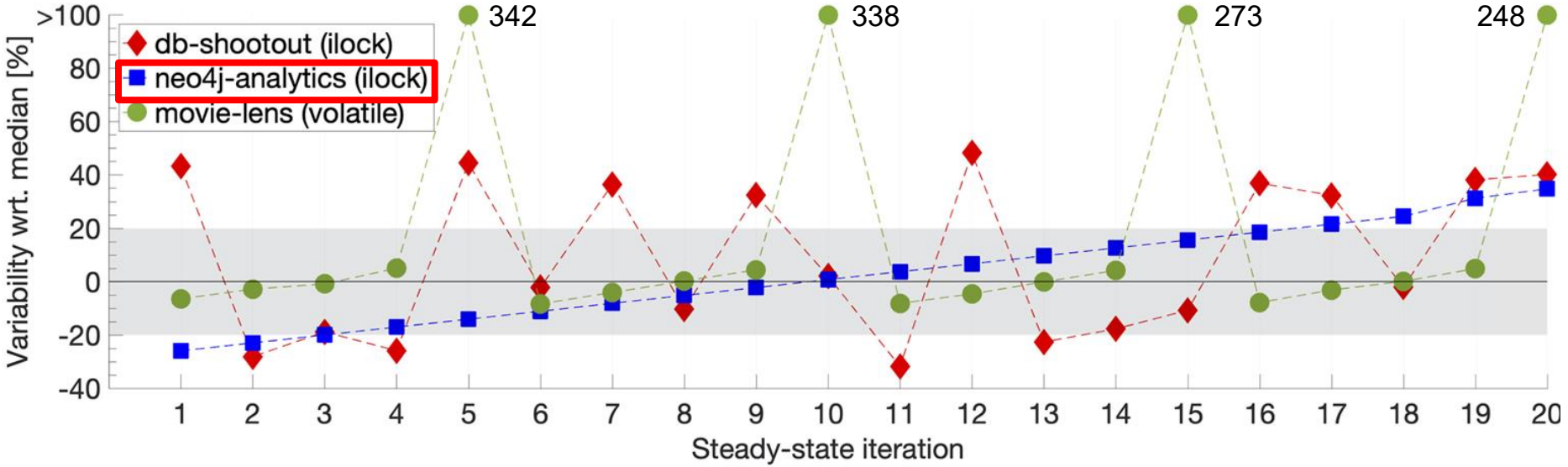
➢ 3 benchmarks show significant variability in a metric

Evaluation – Metric Variability



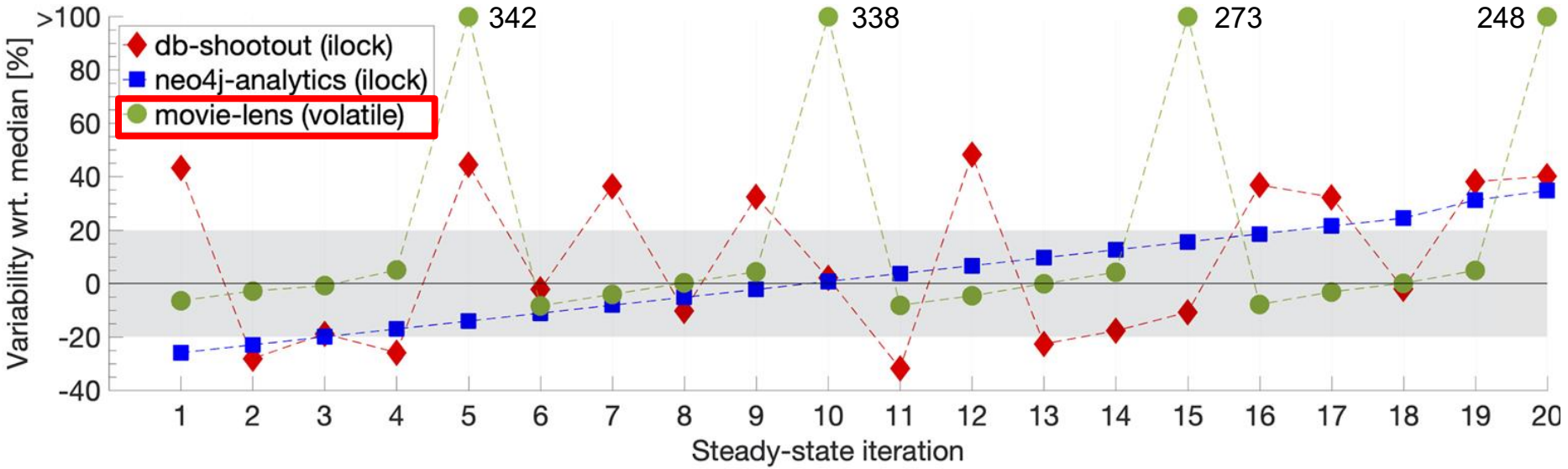
➢ 3 benchmarks show significant variability in a metric

Evaluation – Metric Variability



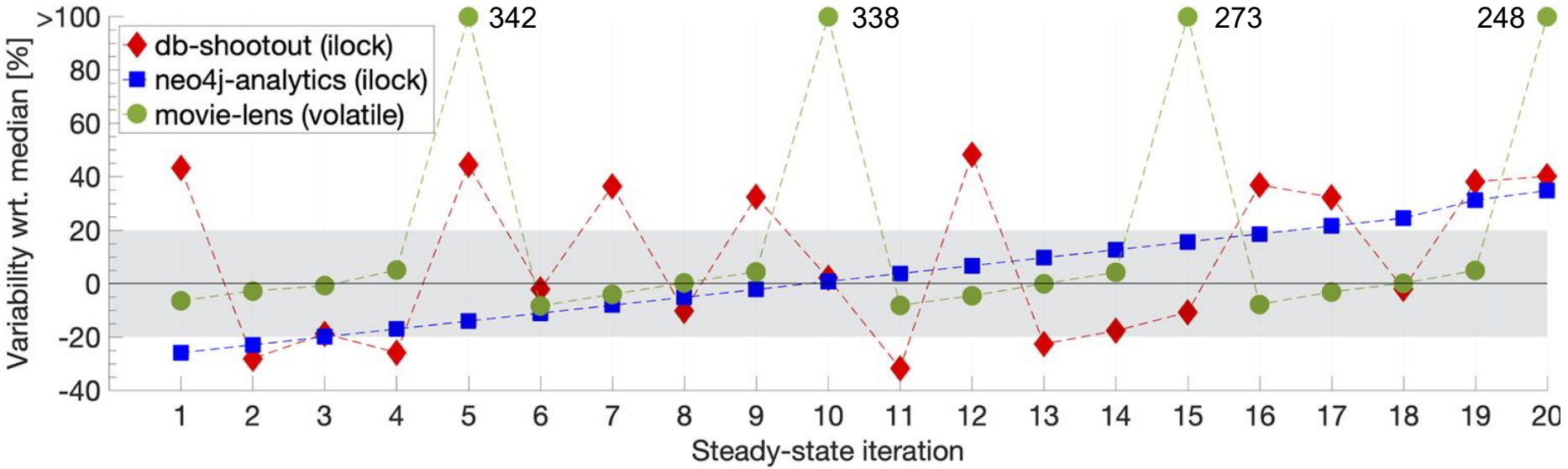
➢ 3 benchmarks show significant variability in a metric

Evaluation – Metric Variability



➢ 3 benchmarks show significant variability in a metric

Evaluation – Metric Variability



- 3 benchmarks show significant variability in a metric
- Patterns indicate occasional or periodic operations that may introduce variability in workloads

Profiling Overhead

Module	OH
thread	1.00
task	1.03
actor	1.01
future	1.01
ilock	1.03
elock	1.01
wait	1.00
join	1.01
park	1.00
synch	1.01
cas	1.01
atomic	1.01
volatile	1.03
scoll	1.00
ccoll	1.01

- Median profiling overhead across all Renaissance benchmarks
  - Measured on 20 steady-state iterations
- Overhead <= 1.01x for most modules
- Overhead = 1.03x for task, ilock and volatile
- Overhead = 1.18x when all modules are active

Limitation

- Over-profiling possible, if JIT compiler removes events of interest without also removing corresponding instrumentation code
  - Well-known limitation of bytecode instrumentation
- Profiling overhead when all modules are active (1.18x) could be significant for some applications
  - Often no need to activate all modules
  - Profiling overhead of individual modules is low

Conclusions

- P^3: a new profiler suite for concurrent applications on the JVM
- Collects many kinds of metrics on parallelism, concurrency and synchronization
- Moderate profiling overhead
- Applicable to prevalent benchmark suites (Renaissance, DaCapo, ScalaBench, SPECjvm2008)
- Suitable for large-scale analysis with NAB
- Fundamental in conducting previous research (e.g., Renaissance)
- P^3 can help researchers conduct novel analyses and better understand multi-threaded applications

Future Work

- Further increase accuracy
- Further decrease profiling overhead
- Expand set of profiled metrics

Thanks for your attention

http://dag.inf.usi.ch/software/p3

- Contacts:

  Andrea Rosà

  andrea.rosa@usi.ch