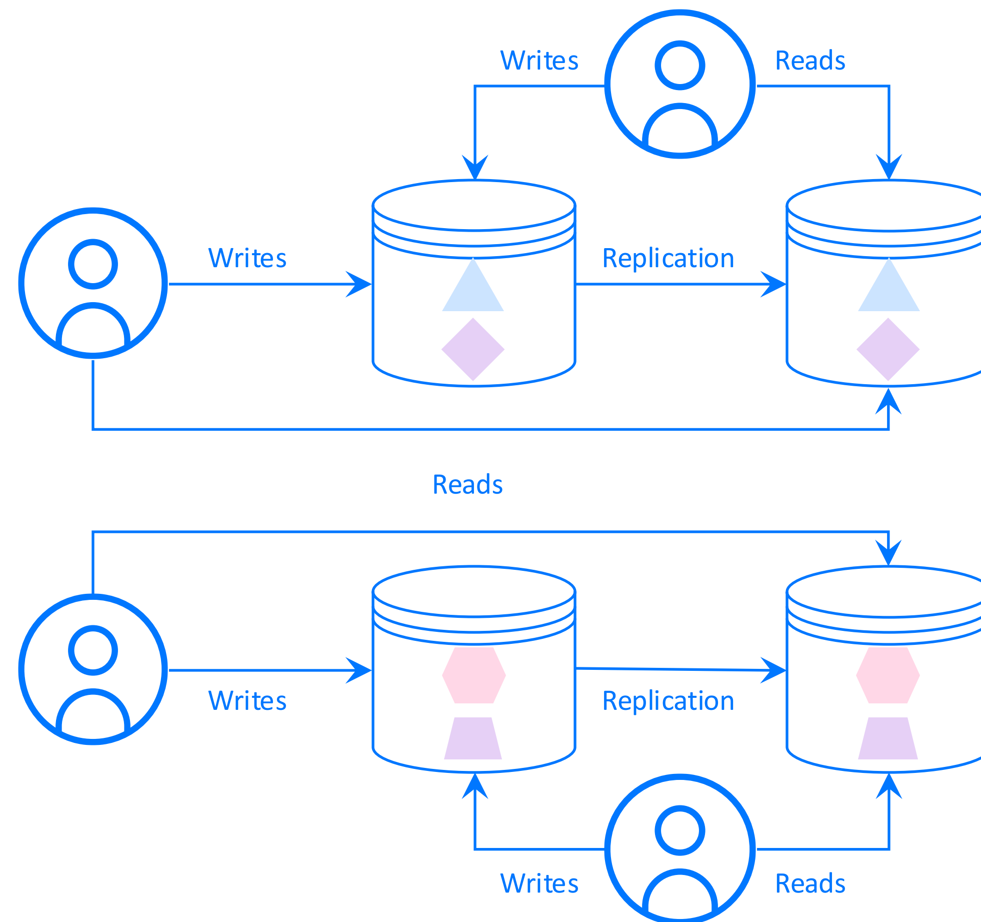
Оптимизация бинарного сериализатора ВКонтакте

Илья Кокорин
i.kokorin@vk.team

Илья Асадуллин
i.asadullin@vkteam

Инфраструктура ВКонтакте: большая распределённая система

- Шардирование для масштабирования
- Репликация для надёжности и масштабирования чтения
- Географически распределённые клиенты
- А ещё CDN, кеши, аналитика, бэкапы…

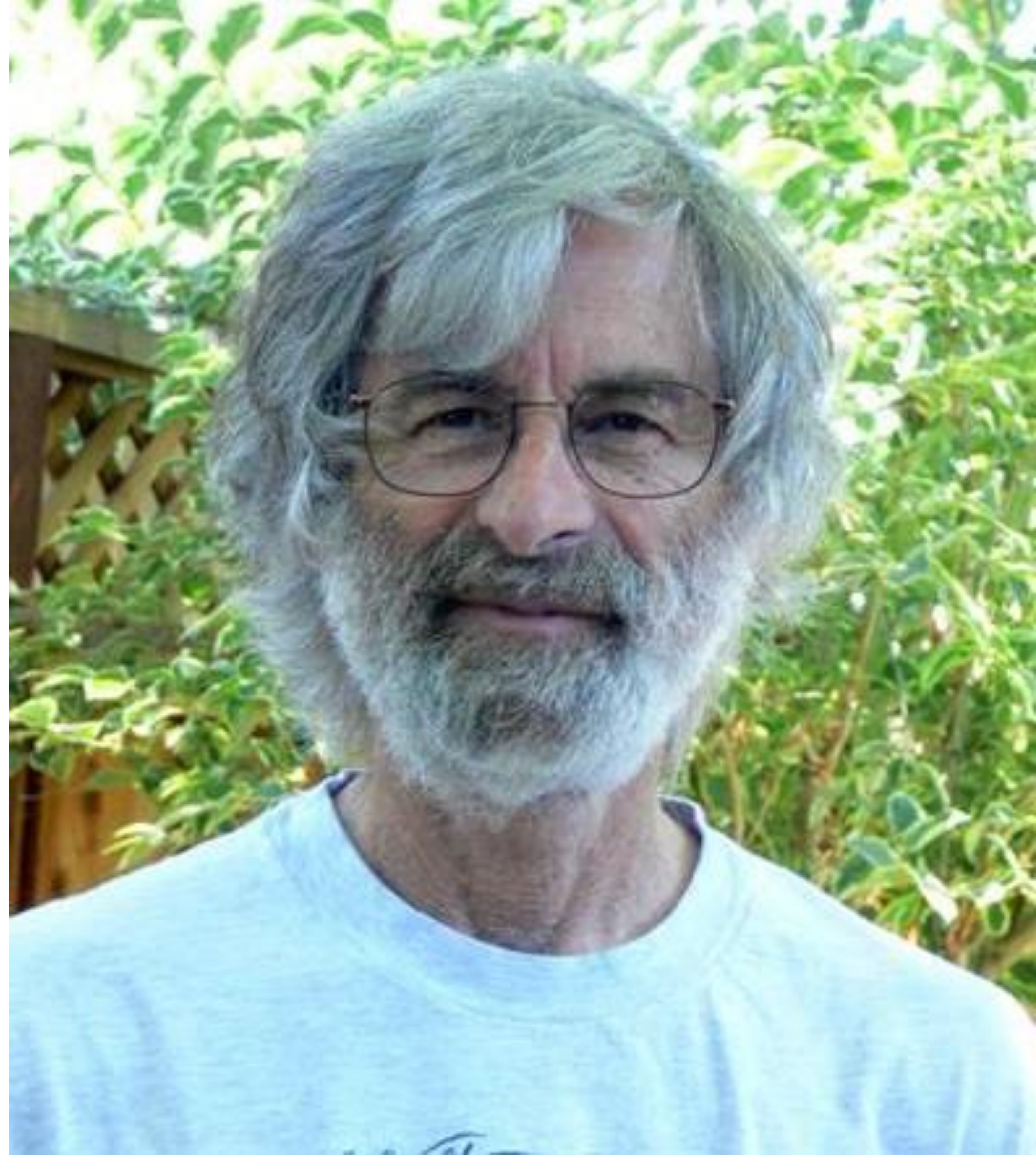


Распределённая система

«

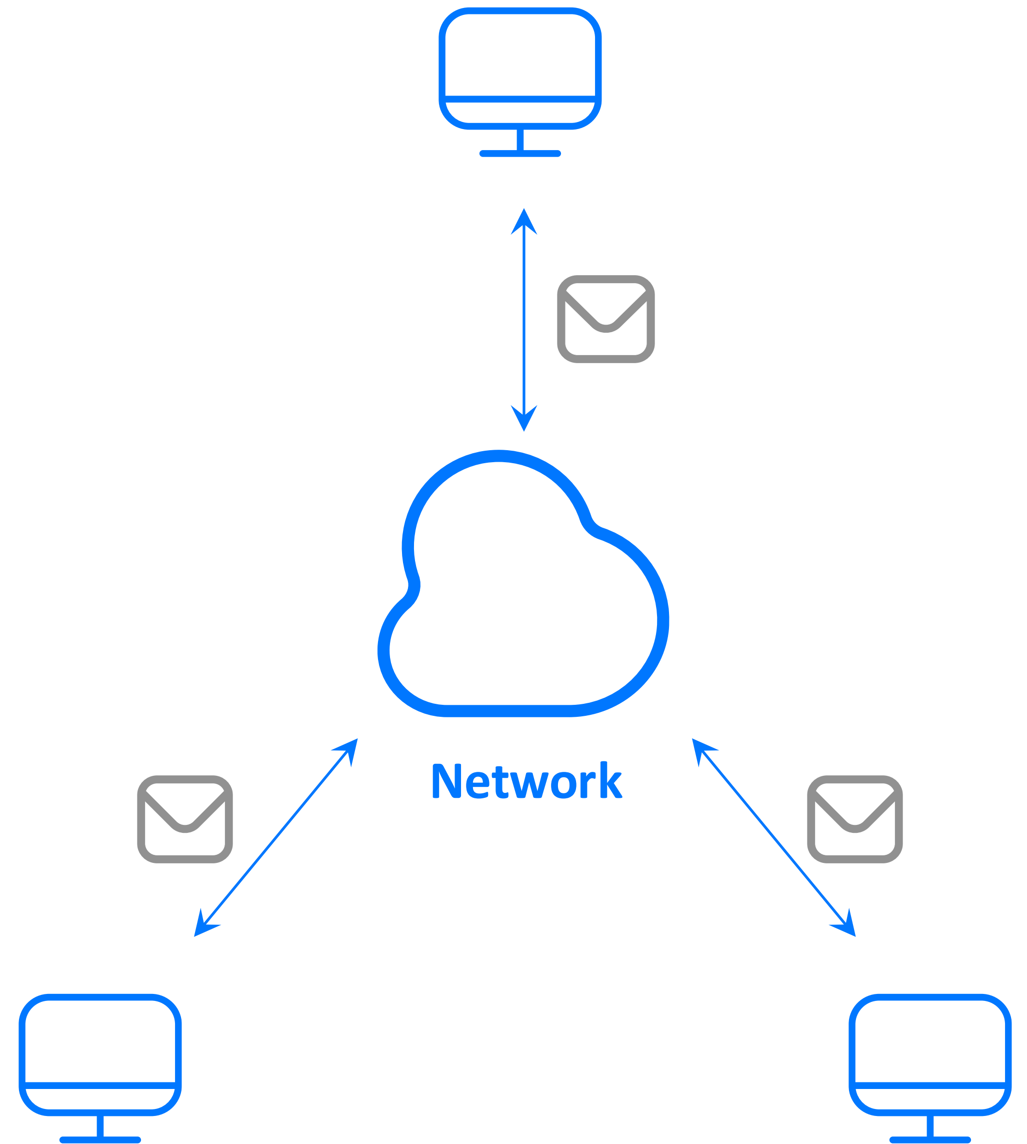
A distributed system is
one in which the failure of
a computer you didn't even know
existed can render your own
computer unusable.

Leslie Lamport



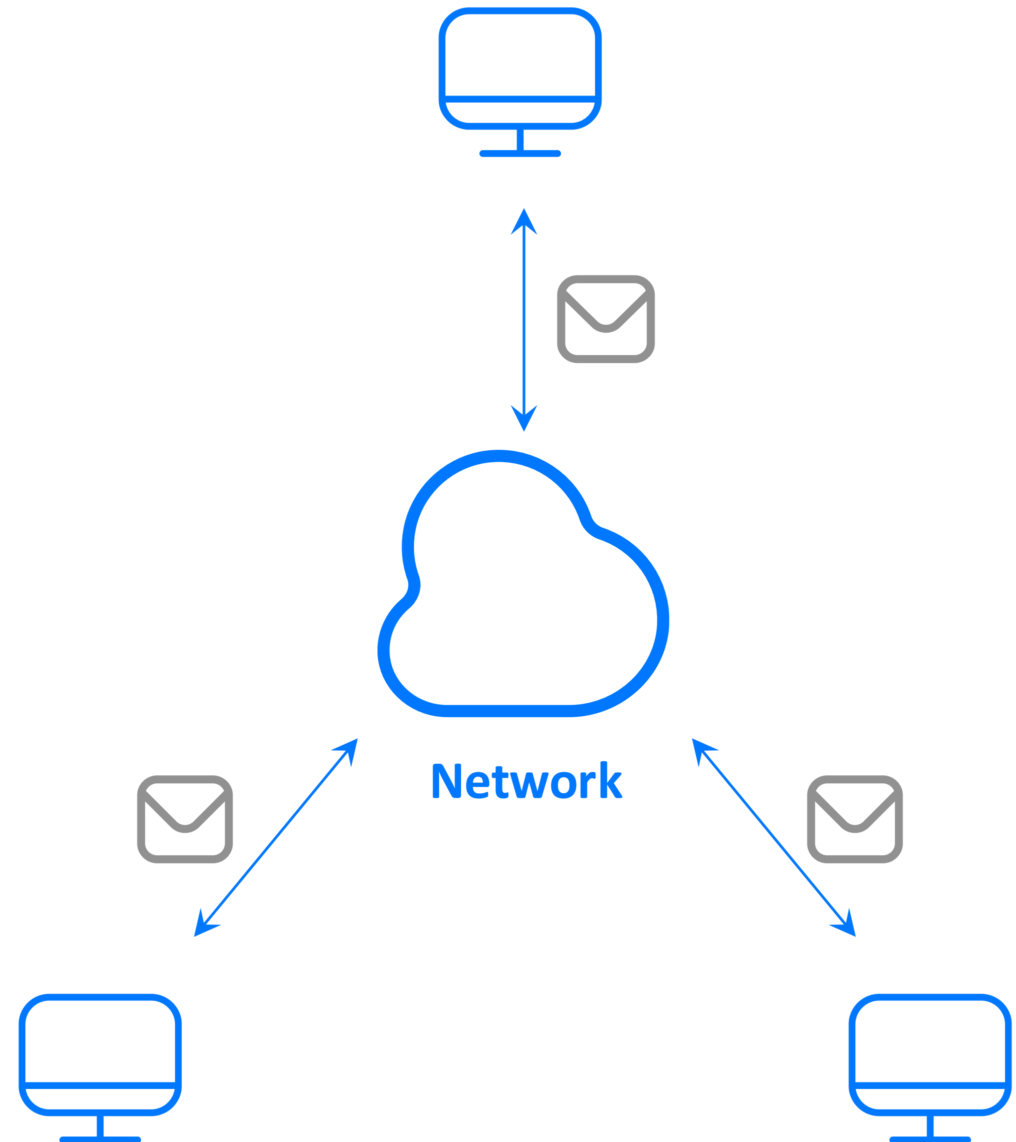
Распределённая система

- Множество решающих общую задачу вычислителей
- У каждого вычислителя независимая память
- Средство коммуникации — отправка сообщений



Распределённая система

- Нет **эффективного** средства передачи сообщений — нет **эффективной** распределённой системы



Примитивы отправки сообщений

API операционной системы позволяет нам отправлять по сети байтовые строки

```
extern ssize_t send (int fd,
                     const void *buf,
                     size_t n,
                     int flags);
```

Примитивы отправки сообщений

API операционной системы позволяет нам отправлять по сети байтовые строки

```
extern ssize_t send (int fd,
                     const void *buf,
                     size_t n,
                     int flags);

extern ssize_t sendto (int fd,
                       const void *buf,
                       size_t n,
                       int flags,
                       CONST_SOCKADDR_ARG addr,
                       socklen_t addr_len);
```

Примитивы отправки сообщений

API операционной системы позволяет нам отправлять по сети байтовые строки

```
extern ssize_t sendmsg (int fd,
                        const struct msghdr *message,
                        int flags);

struct msghdr {
  void      *msg_name;
  socklen_t msg_namelen;

  struct iovec *msg_iov;
  size_t        msg_iovlen;

  void  *msg_control;
  size_t msg_controllen;
  int    msg_flags;
};
```

```
struct iovec {
  void  *iov_base;
  size_t iov_len;
};
```

Примитивы отправки сообщений

API операционной системы позволяет нам отправлять по сети байтовые строки

```
extern int sendmmsg (int fd,
                     struct mmsghdr *vmessages,
                     unsigned int vlen,
                     int flags);
struct msghdr {
  void      *msg_name;
  socklen_t msg_namelen;

  struct iovec *msg_iov;
  size_t       msg_iovlen;

  void  *msg_control;
  size_t msg_controllen;
  int    msg_flags;
};
```

```
struct mmsghdr {
  struct msghdr msg_hdr;
  unsigned int msg_len;
};
struct iovec {
  void  *iov_base;
  size_t iov_len;
};
```

Работа со структурированными данными

При реализации системы мы хотим оперировать не байтовыми строками, а доменными объектами

Работа со структурированными данными

Описываем тип доменного объекта на специальном языке описания сущностей

```
public = Visibility;
friendsOnly = Visibility;

newPost
  user_id:          long
  text:             string
  visibility:       Visibility
  attachments_urls: vector url
= NewPost;
```

Получаем код для чтения сущности из байтовой строки и обращения к её полям…

```
enum class visibility_t {
  FRIENDS_ONLY,
  EVERYBODY,
};

struct new_post_t {
  int64_t get_user_id();
  std::string_view get_text();
  visibility_t get_visibility();
  std::span<url> get_attachments_urls();

  void fetch(byte_source_t&);
};
```

Работа со структурированными данными

Описываем тип доменного объекта на специальном языке описания сущностей

```
public = Visibility;
friendsOnly = Visibility;

newPost
  user_id:          long
  text:             string
  visibility:       Visibility
  attachments_urls: vector url
= NewPost;
```

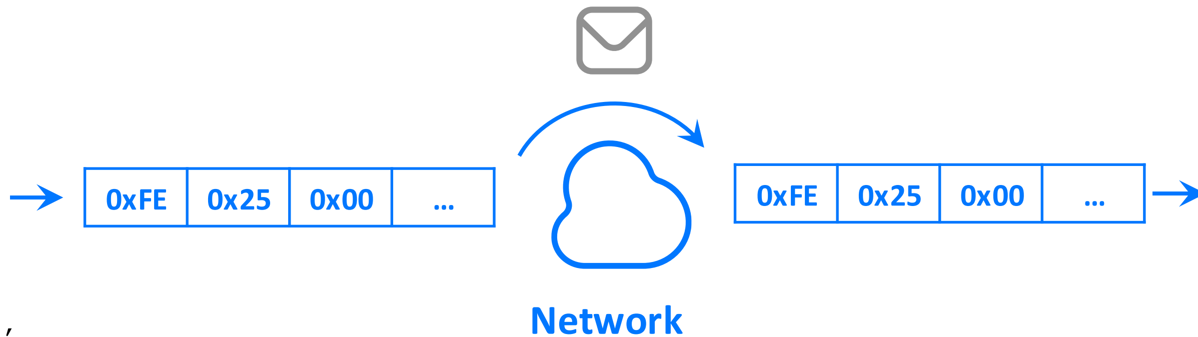
… и для сохранения сущности в байтовую строку

```
enum class visibility_t {
  FRIENDS_ONLY,
  EVERYBODY,
};

struct new_post_storer_t {
  void store_user_id(int64_t);
  void store_text(std::string_view);
  void store_visibility(visibility_t);
  void store_attachments_urls(std::span<url>);

  explicit new_post_storer_t(byte_source&)
  ~new_post_storer_t();
};
```

Работа со структурированными данными

- Превращаем сложный структурированный объект в байтовую строку
- Посылаем байтовую строку по сети
- Из байтовой строки парсим объект, идентичный исходному

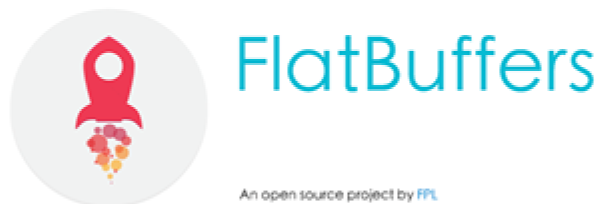
```
NewPost {
    user_id: 473881819,
    text: "Hello, world! ",
    visibility: FRIENDS_ONLY,
    attachments: [
        "some/image/url.png",
        "anohter/ image/url.png",
    ]
}
```

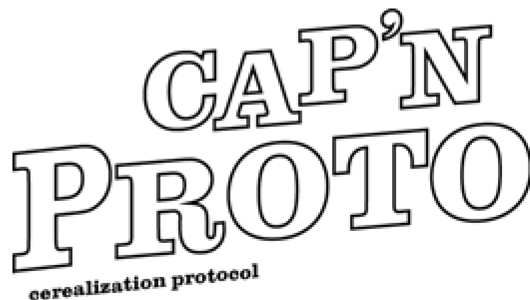



```
NewPost {
    user_id: 473881819,
    text: "Hello, world! ",
    visibility: FRIENDS_ONLY,
    attachments: [
        "some/image/url.png",
        "anohter/ image/url.png",
    ]
}
```

Работа со структурированными данными

Известны многие решения этой задачи…





MessagePack

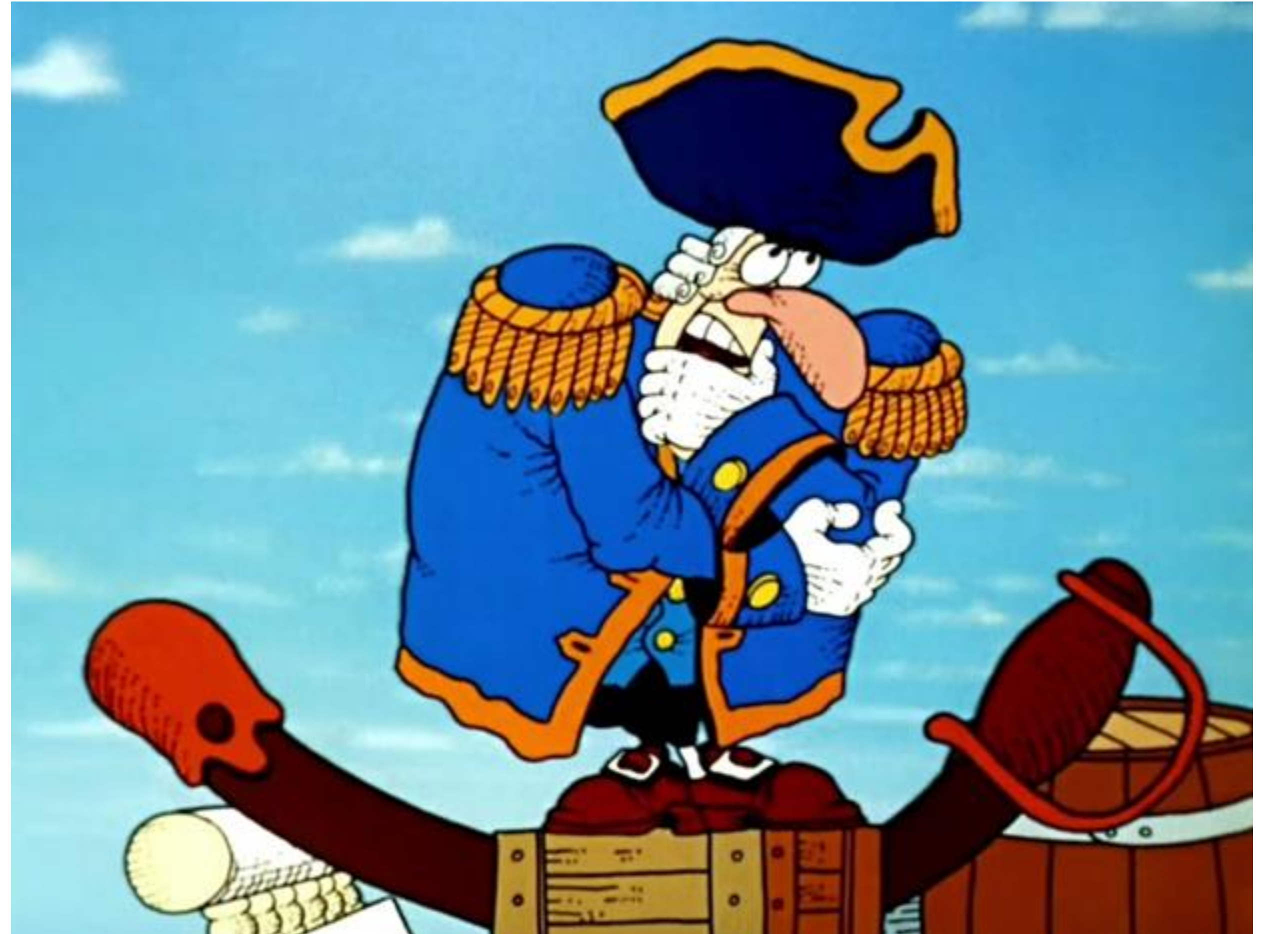
It's like JSON.
but fast and small.

Работа со структурированными данными

… но мы

написали своё

Зачем?



Насколько быстрее? TL vs Protobuf

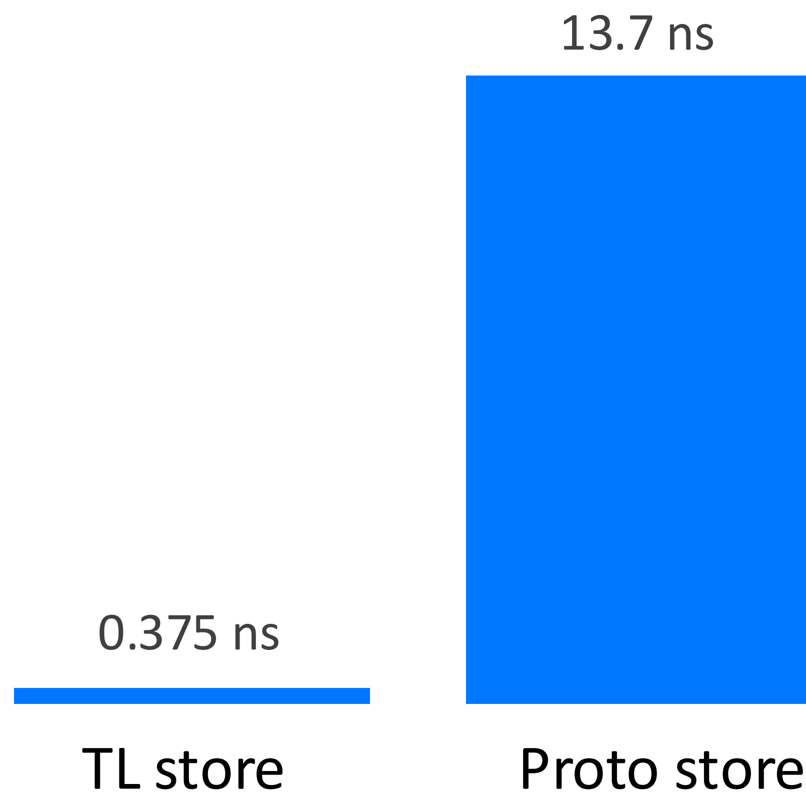
```
simple
   a: int
   b: int
   c: int
   d: int
   e: int
= Simple;
```

```
static void BM_TL_simple_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<int32_t> dist{1, 100};
    tl_simple_t from{.a = dist(rng), .b = dist(rng), .c = dist(rng), .d = dist(rng), .e = dist(rng)};
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = from.tl_store(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_simple_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<int32_t> dist{1, 100};
    protobench::Simple from;
    from.set_a(dist(rng));
    from.set_b(dist(rng));
    from.set_c(dist(rng));
    from.set_d(dist(rng));
    from.set_e(dist(rng));
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        from.SerializeToArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}
```

```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
}
```

Насколько быстрее? TL vs Protobuf

```
simple
  a: int
  b: int
  c: int
  d: int
  e: int
= Simple;
```




```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
}
```

Насколько быстрее? TL vs Protobuf

```
simple
   a: int
   b: int
   c: int
   d: int
   e: int
= Simple;
```

```
static void BM_TL_simple_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<int32_t> dist{1, 100};
    tl_simple_t from{.a = dist(rng), .b = dist(rng), .c = dist(rng), .d = dist(rng), .e = dist(rng)};
    std::array<uint8_t, 4096> buf{};
    std::ignore = from.tl_store(buf.data(), buf.size());
    tl_simple_t to{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = to.tl_fetch(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_simple_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<int32_t> dist{1, 100};
    protobench::Simple from;
    from.set_a(dist(rng));
    from.set_b(dist(rng));
    from.set_c(dist(rng));
    from.set_d(dist(rng));
    from.set_e(dist(rng));
    std::array<uint8_t, 4096> buf{};
    std::ignore = from.SerializeToArray(buf.data(), buf.size());
    protobench::Simple to;
    for ([[maybe_unused]] const auto _ : state) {
        to.ParseFromArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}
```

```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
}
```

Насколько быстрее? TL vs Protobuf

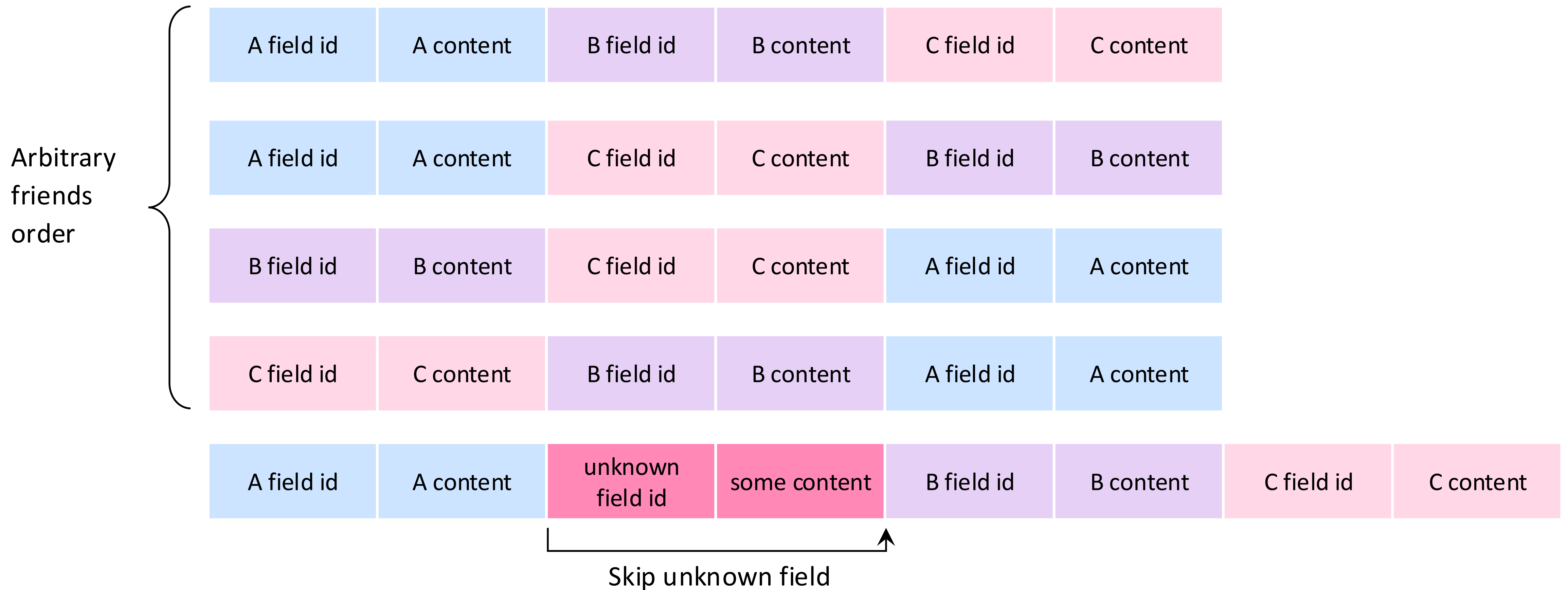
```
simple
  a: int
  b: int
  c: int
  d: int
  e: int
= Simple;
```

	TL parse	Proto parse
Время	0,457 ns	24,6 ns

```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
}
```

Насколько быстрее? TL vs Protobuf

Почему protobuf так сильно проигрывает?



Насколько быстрее?
TL vs Protobuf

Читаем ID поля

```
for (;;) {
  std::pair<uint32_t, bool> p = input->ReadTagWithCutoffNoLastTag(127u);
  tag = p.first;
  if (!p.second) goto handle_unusual;

  switch (GetTagFieldNumber(tag)) {
    case 1: { // int32 a = 1;
      if (tag == 8u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &a)));
      } else goto handle_unusual;
      break;
    }

    case 2: { // int32 b = 2;
      if (tag == 16u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &b)));
      } else goto handle_unusual;
      break;
    }

    /* cases for reading c, d, and e */

    [[unlikely]] default: {
      handle_unusual:
        if (tag == 0) goto success;
        DO_(SkipField(input, tag,
                      _internal_metadata.mutable_unknown_fields()));
        break;
    }
  }
}
```

Насколько быстрее? TL vs Protobuf

Понимаем по ID, что это за поле

Парсим данные этого поля

```
for (;;) {
  std::pair<uint32_t, bool> p = input->ReadTagWithCutoffNoLastTag(127u);
  tag = p.first;
  if (!p.second) goto handle_unusual;

  switch (GetTagFieldNumber(tag)) {
    case 1: { // int32 a = 1;
      if (tag == 8u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &a)));
      } else goto handle_unusual;
      break;
    }

    case 2: { // int32 b = 2;
      if (tag == 16u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &b)));
      } else goto handle_unusual;
      break;
    }

    /* cases for reading c, d, and e */

    [[unlikely]] default: {
      handle_unusual:
        if (tag == 0) goto success;
        DO_(SkipField(input, tag,
                      _internal_metadata.mutable_unknown_fields()));
        break;
    }
  }
}
```

Насколько быстрее? TL vs Protobuf

Понимаем по ID, что это за поле

Парсим данные этого поля

```
for (;;) {
  std::pair<uint32_t, bool> p = input->ReadTagWithCutoffNoLastTag(127u);
  tag = p.first;
  if (!p.second) goto handle_unusual;

  switch (GetTagFieldNumber(tag)) {
    case 1: { // int32 a = 1;
      if (tag == 8u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &a)));
      } else goto handle_unusual;
      break;
    }

    case 2: { // int32 b = 2;
      if (tag == 16u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &b)));
      } else goto handle_unusual;
      break;
    }

    /* cases for reading c, d, and e */

    [[unlikely]] default: {
      handle_unusual:
        if (tag == 0) goto success;
        DO_(SkipField(input, tag,
                       _internal_metadata.mutable_unknown_fields()));
        break;
    }
  }
}
```

Насколько быстрее? TL vs Protobuf

Неизвестные поля пропускаем

```
for (;;) {
  std::pair<uint32_t, bool> p = input->ReadTagWithCutoffNoLastTag(127u);
  tag = p.first;
  if (!p.second) goto handle_unusual;

  switch (GetTagFieldNumber(tag)) {
    case 1: { // int32 a = 1;
      if (tag == 8u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &a)));
      } else goto handle_unusual;
      break;
    }

    case 2: { // int32 b = 2;
      if (tag == 16u) {
        DO_((ReadPrimitive<int32_t, TYPE_INT32>(input, &b)));
      } else goto handle_unusual;
      break;
    }

    /* cases for reading c, d, and e */

    [[unlikely]] default: {
      handle_unusual:
        if (tag == 0) goto success;
        DO_(SkipField(input, tag,
                      _internal_metadata.mutable_unknown_fields()));
        break;
    }
  }
}
```

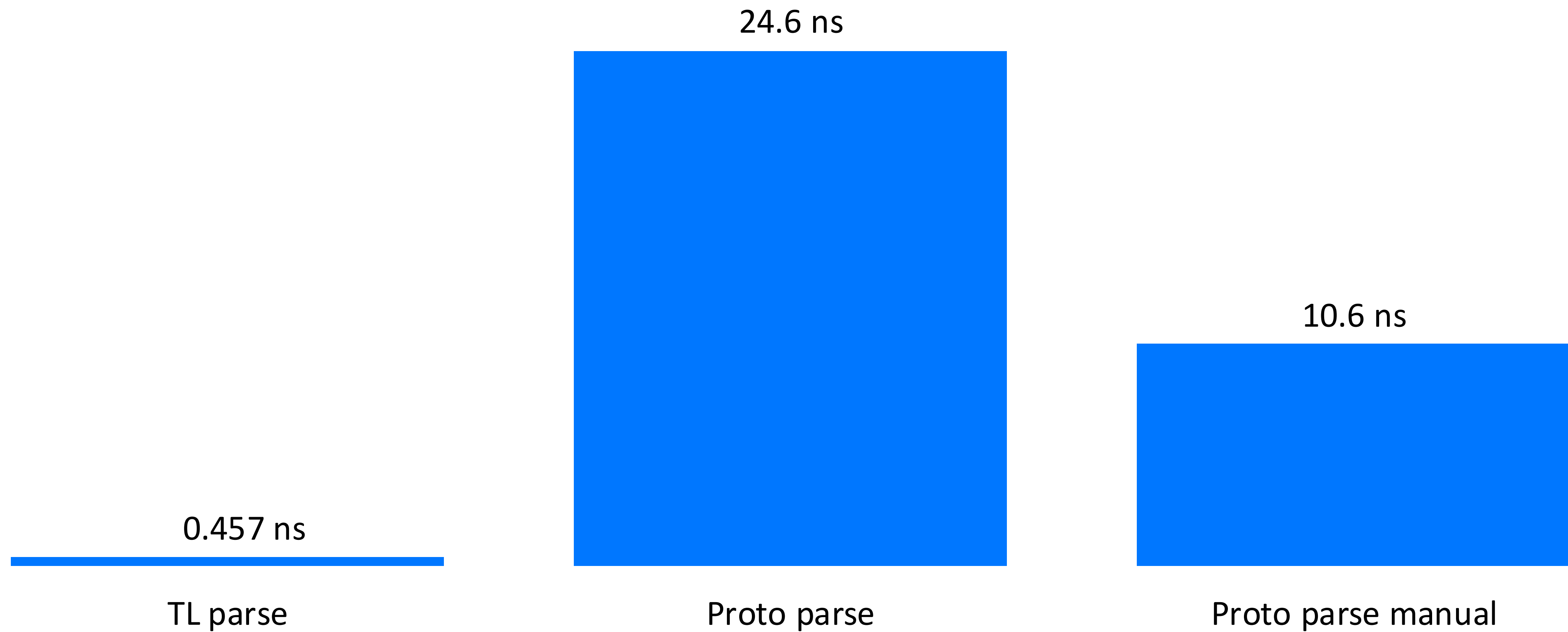
Насколько быстрее?
TL vs Protobuf

Оптимизируем protobuf вручную

Избавляемся от цикла, предполагая что поля записаны в порядке a-b-c-d-e

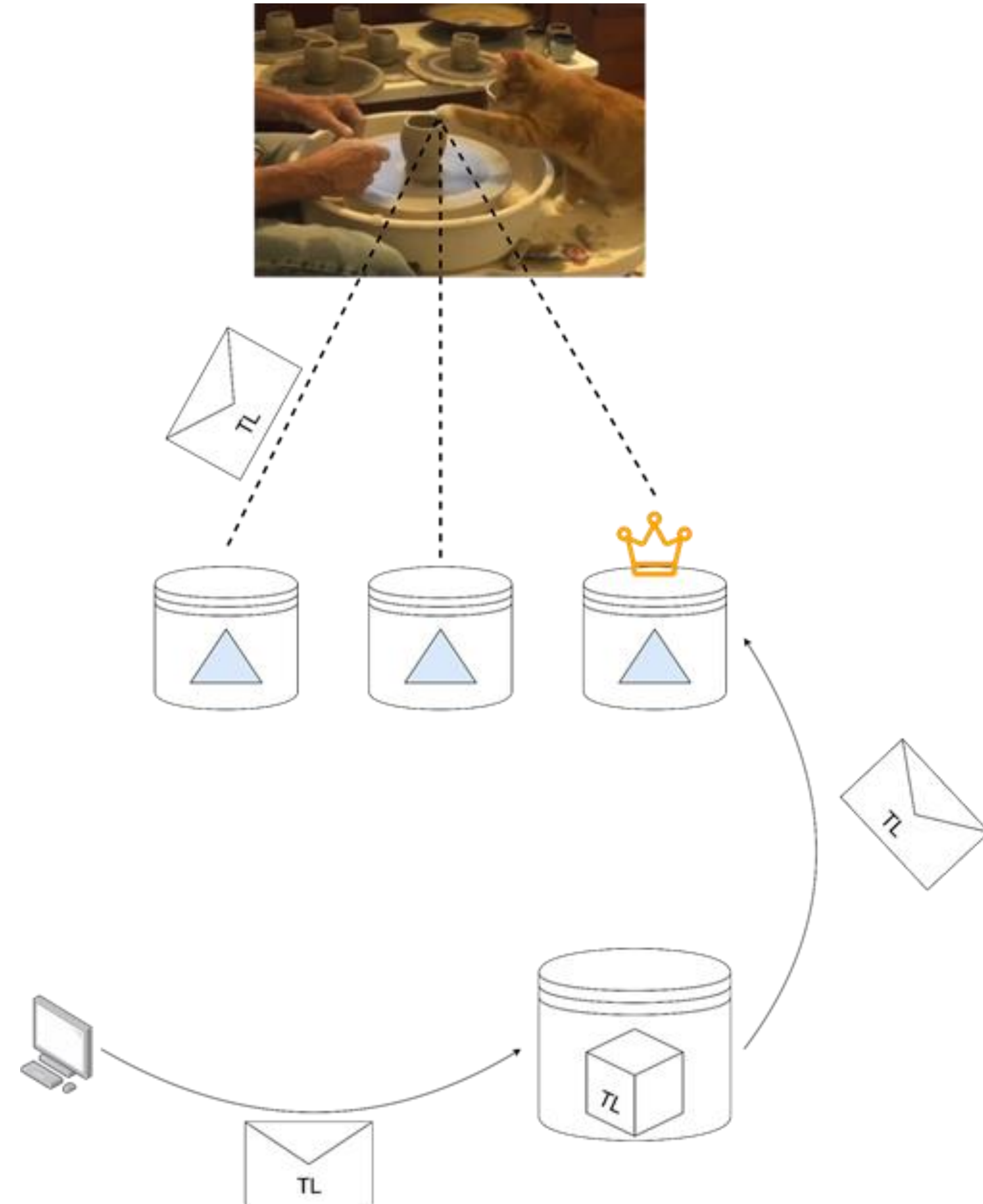
```
auto tag = input.ReadTagNoLastTag();
if (tag != 8u) [[unlikely]] std::abort();
if (!ReadPrimitive<int32_t, TYPE_INT32>(&input, &to.a))
    [[unlikely]] std::abort();

tag = input.ReadTagNoLastTag();
if (tag != 16u) [[unlikely]] std::abort();
if (!ReadPrimitive<int32_t, TYPE_INT32>(&input, &to.b))
    [[unlikely]] std::abort();

tag = input.ReadTagNoLastTag();
if (tag != 24u) [[unlikely]] std::abort();
if (!ReadPrimitive<int32_t, TYPE_INT32>(&input, &to.c))
    [[unlikely]] std::abort();

tag = input.ReadTagNoLastTag();
if (tag != 32u) [[unlikely]] std::abort();
if (!ReadPrimitive<int32_t, TYPE_INT32>(&input, &to.d))
    [[unlikely]] std::abort();

tag = input.ReadTagNoLastTag();
if (tag != 40u) [[unlikely]] std::abort();
if (!ReadPrimitive<int32_t, TYPE_INT32>(&input, &to.e))
    [[unlikely]] std::abort();
```

Насколько быстрее? TL vs Protobuf



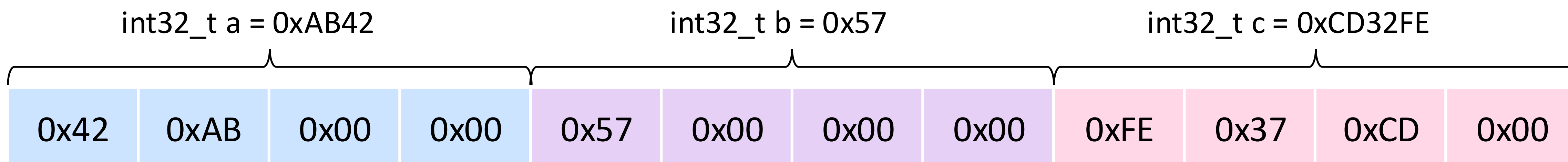
Использование TL во ВКонтакте

- Общение между клиентами и базами данных;
- Общение баз данных между собой
- Храним на диске как часть персистентного состояния;
- Используем в протоколе консенсусной репликации;
- …
- Роль TL в инфраструктуре ВКонтакте [схожа] с ролью Protobuf в инфраструктуре Google



Структура TL: поля примитивных типов

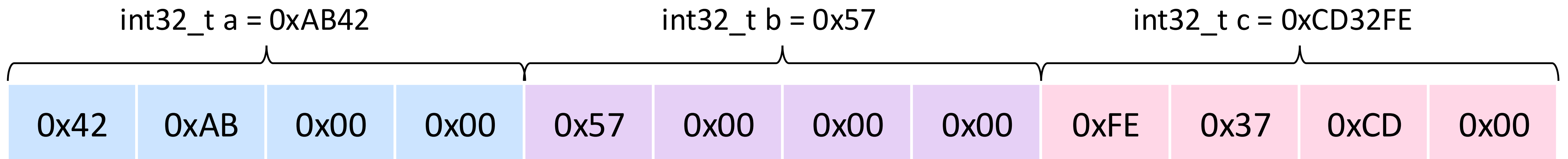
```
simple a: int b: int c: int = Simple
```



- Значения полей идут один за другим в порядке их указания в схеме
- Значения полей примитивного типа в little endian

Структура TL: поля примитивных типов

```
simple a: int b: int c: int = Simple
```



- Layout байтовой строки TL совпадает с layout POD-структуры
- (De)-serialization as fast as memcpy

```
struct simple_pod_t {
  int32_t a;
  int32_t b;
  int32_t c;
};
```

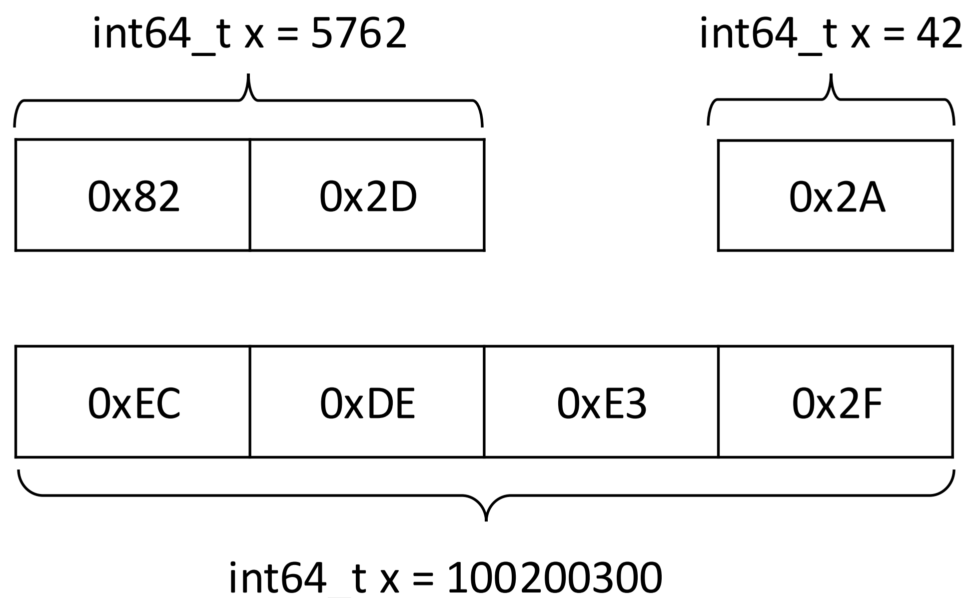
Varint в Protobuf

Целые числа в Protobuf
записываются в кодировке varint

- Идентификаторы полей тоже

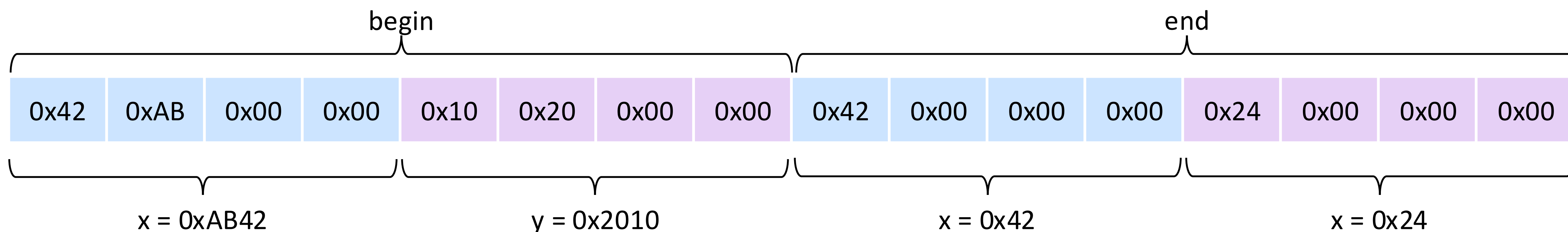
Длина записи числа зависит
от количества значимых битов

- А не от размера типа (`int8_t`, `int16_t`, `int32_t`, `int64_t`)



Структура TL: вложенные структуры

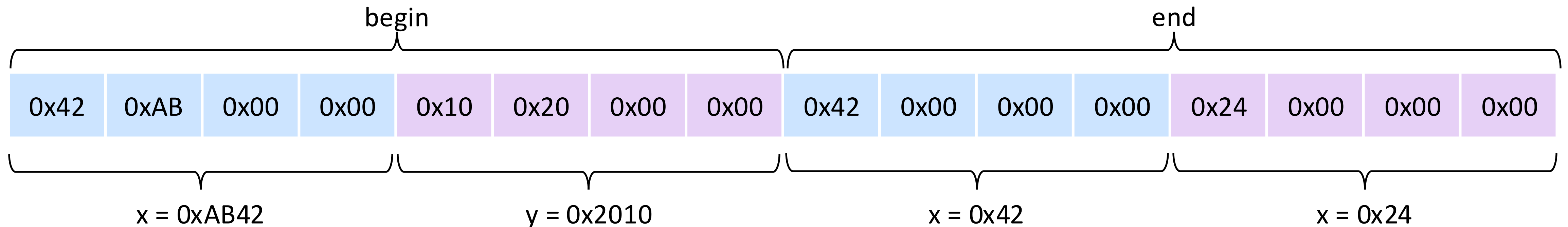
```
point x: int y: int = Point
segment begin: point end: point = Segment
```



Сериализованные структуры идут друг за другом, без служебной информации

Структура TL: вложенные структуры

```
point x: int y: int = Point
segment begin: point end: point = Segment
```



Layout такой же,
как у POD-структур

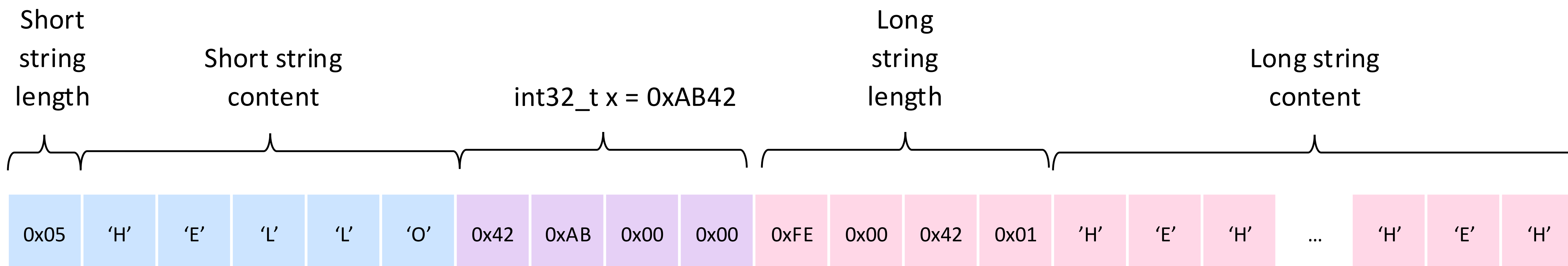
```
struct point_t {
  int32_t x;
  int32_t y;
};
```

```
struct segment_t {
  point_t begin;
  point_t end;
};
```

Строки в TL

- Сначала записывается её длина, потом её байты
- Есть оптимизация для краткой записи длин коротких строк
- В отличие от записи маленьких чисел, потому что коротких строк сильно больше

```
data
  short_string: string
  x:            int
  long_string:  string
= Data;
```



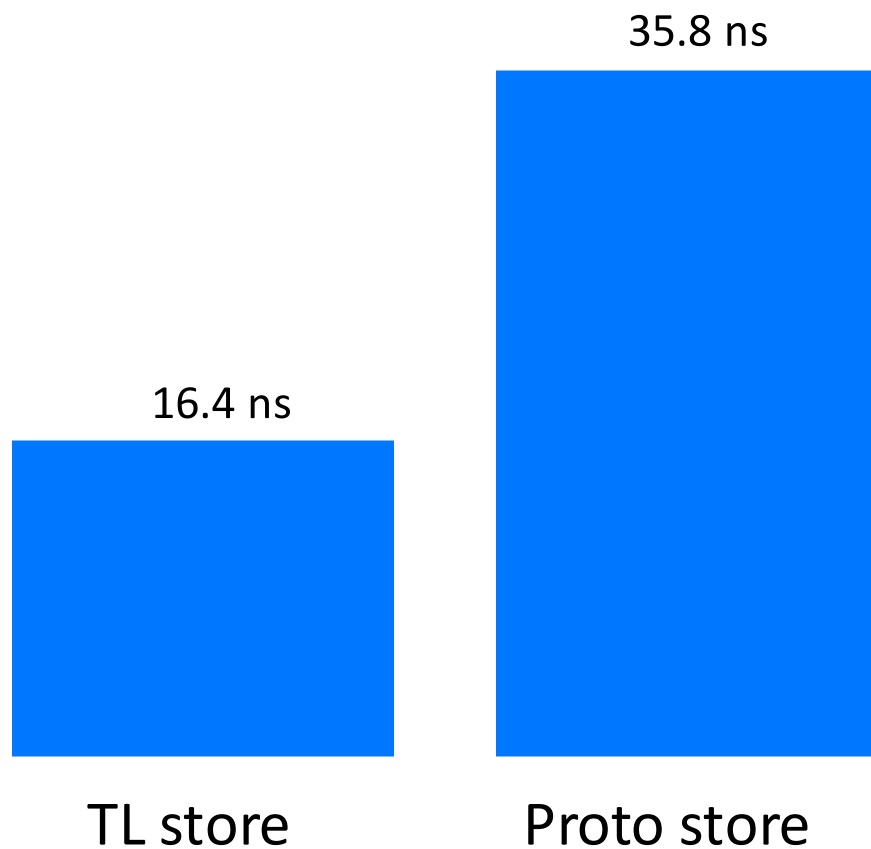
Бенчмарки работы со строками

```cpp
static void BM_TL_strings_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    tl_strings_t from{.short_string = std::string(10, dist(rng)), .long_string = std::string(1000, dist(rng))};
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = from.tl_store(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_strings_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    protobench::Strings from;
    from.set_short_string(std::string(10, dist(rng)));
    from.set_long_string(std::string(1000, dist(rng)));
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        from.SerializeToArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}
```

```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

Бенчмарки работы со строками




```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

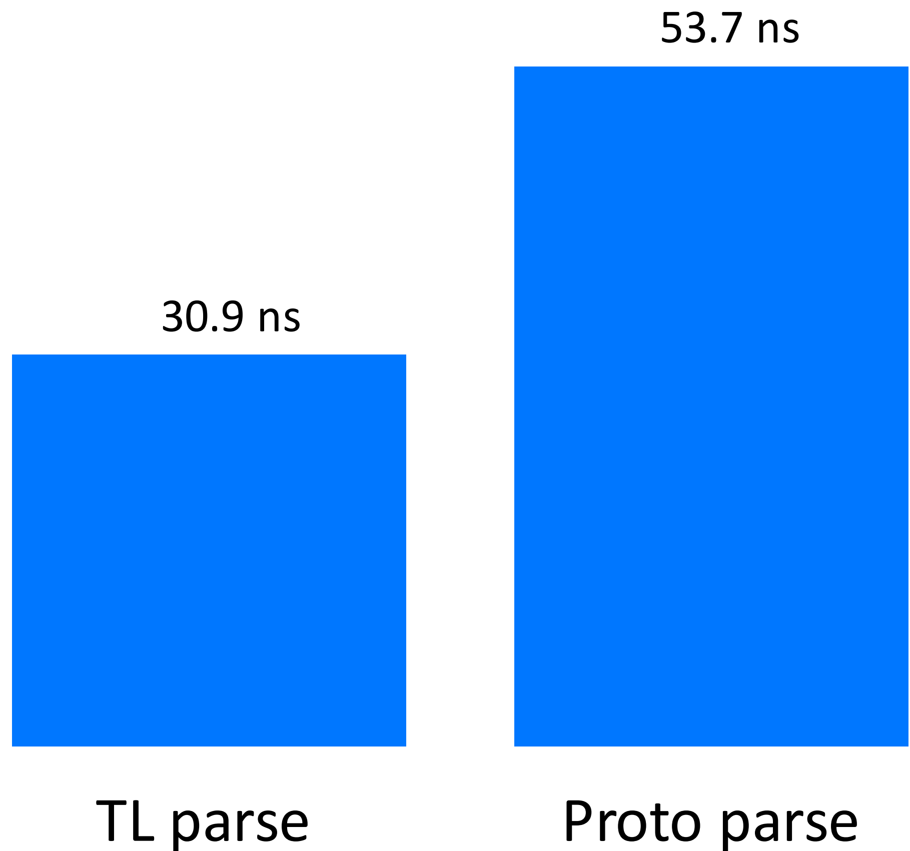
Бенчмарки работы со строками

```cpp
static void BM_TL_strings_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    tl_strings_t from{.short_string = std::string(10, dist(rng)), .long_string = std::string(1000, dist(rng))};
    std::array<uint8_t, 4096> buf{};
    std::ignore = from.tl_store(buf.data(), buf.size());
    tl_strings_t to{};
    to.short_string.reserve(4096);
    to.long_string.reserve(4096);
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = to.tl_fetch(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_strings_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    protobench::Strings from{};
    from.set_short_string(std::string(10, dist(rng)));
    from.set_long_string(std::string(1000, dist(rng)));
    std::array<uint8_t, 4096> buf{};
    from.SerializeToArray(buf.data(), buf.size());
    protobench::Strings to{};
    to.mutable_short_string()->reserve(4096);
    to.mutable_long_string()->reserve(4096);
    for ([[maybe_unused]] const auto _ : state) {
        to.ParseFromArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}
```

```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

Бенчмарки работы со строками




```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

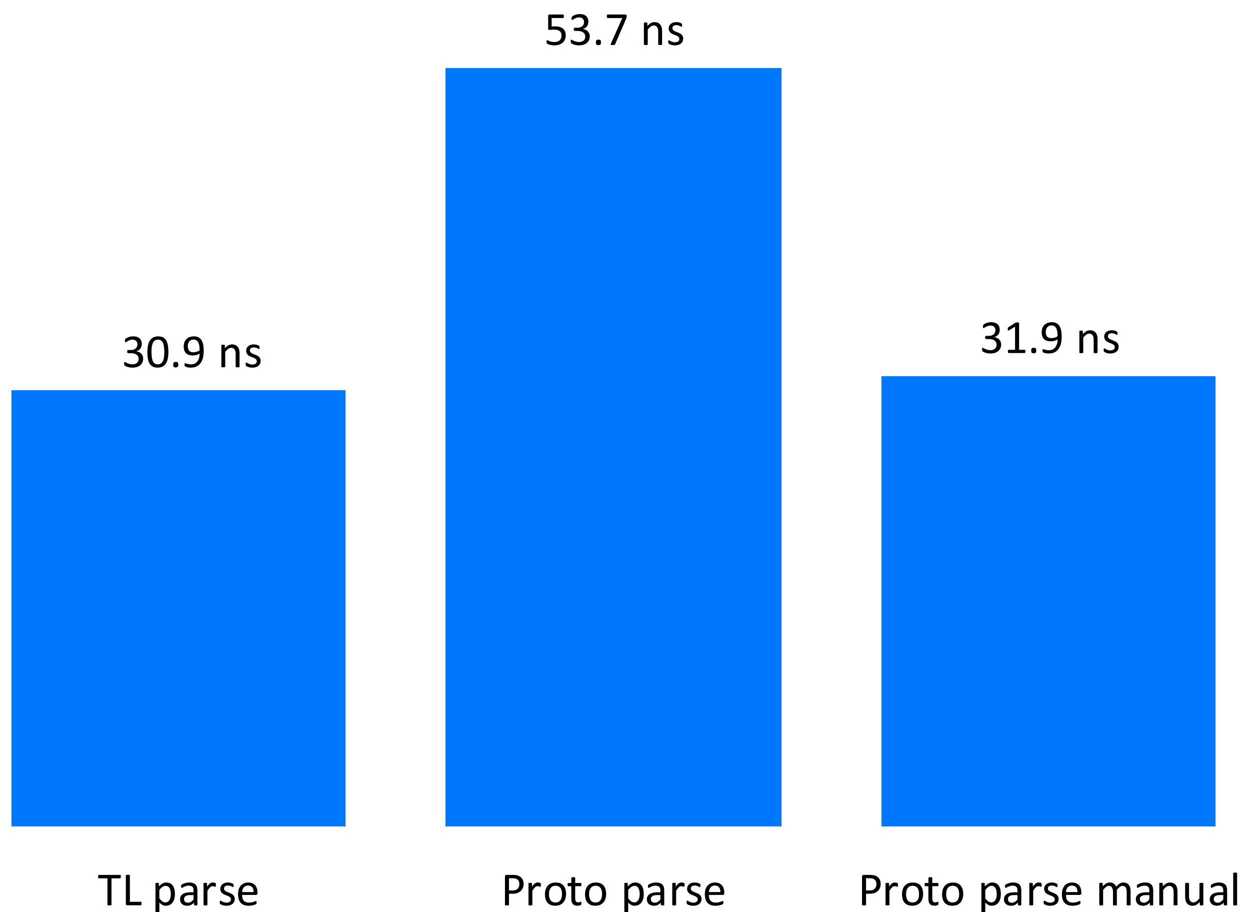
Бенчмарки работы со строками

```
auto tag = input.ReadTagNoLastTag();
if (tag != 10u) [[unlikely]] std::abort();
if (!ReadString(&input, &to.short_string))
    [[unlikely]] std::abort();

tag = input.ReadTagNoLastTag();
if (tag != 18u) [[unlikely]] std::abort();
if (!ReadString(&input, &to.long_string))
    [[unlikely]] std::abort();
```

```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

Бенчмарки работы со строками




```
strings
  short_string: string
  long_string: string
= Strings;
```

```
message Strings {
  bytes short_string = 1;
  bytes long_string = 2;
}
```

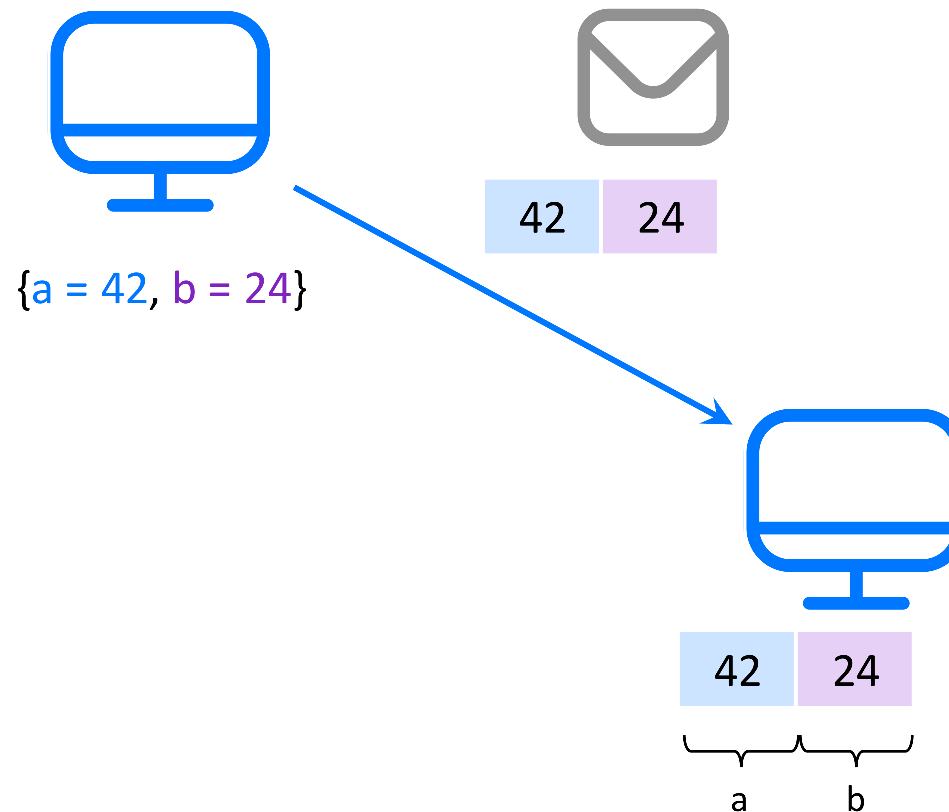
Обновление схемы

Хотим добавить поле в структуру

```
data x: int y: int = Data
```

Получить

```
data x: int y: int c: int = Data
```



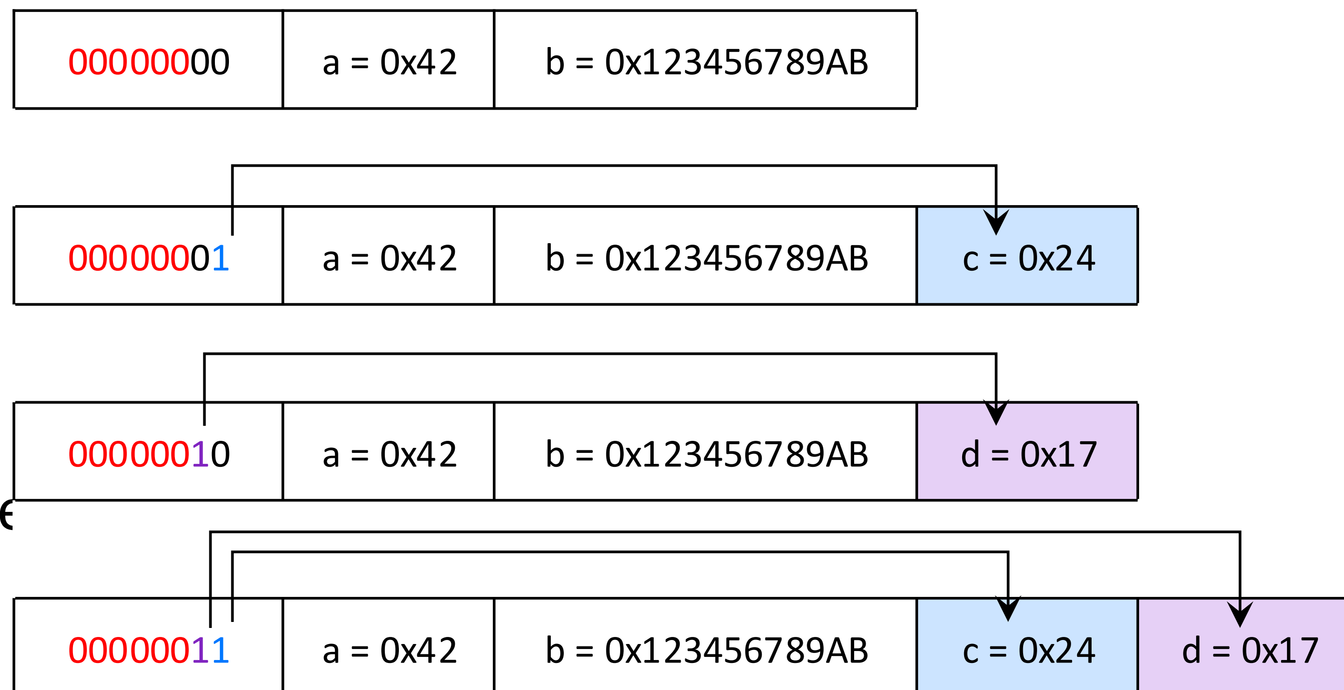
Маски полей

- Поле `fields_mask` для обеспечения расширяемости
- Поле `c` задано если у `fields_mask` нулевой бит равен единице
- Поле `d` задано если у `fields_mask` первый бит равен единице

```
data
  fields_mask: #
  a:           int
  b:           long
  c:           mask.0?int
  d:           mask.1?int
= Data;
```

Маски полей

- Поле `c` задано если у `fields_mask` нулевой бит равен единице
- Поле `d` задано если у `fields_mask` первый бит равен единице



Маски полей: сериализация

При записи запоминаем указатель на место записи маски полей

Зануляем маску

```
if (size < sizeof(uint8_t) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
uint8_t* mask_ptr = buf;
*mask_ptr = 0;
memcpy(buf + sizeof(uint8_t), &a, sizeof(a));
memcpy(buf + sizeof(uint8_t) + sizeof(a), &b, sizeof(b));
buf += sizeof(uint8_t) + sizeof(a) + sizeof(b);
size -= sizeof(uint8_t) + sizeof(a) + sizeof(b);
if (c.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*c, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 0);
}
if (d.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*d, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 1);
}
```

Маски полей: сериализация

Безусловно присутствующие поля сериализуем как обычно

```
if (size < sizeof(uint8_t) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
uint8_t* mask_ptr = buf;
*mask_ptr = 0;
memcpy(buf + sizeof(uint8_t), &a, sizeof(a));
memcpy(buf + sizeof(uint8_t) + sizeof(a), &b, sizeof(b));
buf += sizeof(uint8_t) + sizeof(a) + sizeof(b);
size -= sizeof(uint8_t) + sizeof(a) + sizeof(b);
if (c.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*c, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 0);
}
if (d.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*d, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 1);
}
```

Маски полей: сериализация

При записи поля добавляем биты к маске

```
if (size < sizeof(uint8_t) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
uint8_t* mask_ptr = buf;
*mask_ptr = 0;
memcpy(buf + sizeof(uint8_t), &a, sizeof(a));
memcpy(buf + sizeof(uint8_t) + sizeof(a), &b, sizeof(b));
buf += sizeof(uint8_t) + sizeof(a) + sizeof(b);
size -= sizeof(uint8_t) + sizeof(a) + sizeof(b);
if (c.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*c, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 0);
}
if (d.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*d, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 1);
}
```

Маски полей: сериализация

При записи поля добавляем биты к маске

```
if (size < sizeof(uint8_t) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
uint8_t* mask_ptr = buf;
*mask_ptr = 0;
memcpy(buf + sizeof(uint8_t), &a, sizeof(a));
memcpy(buf + sizeof(uint8_t) + sizeof(a), &b, sizeof(b));
buf += sizeof(uint8_t) + sizeof(a) + sizeof(b);
size -= sizeof(uint8_t) + sizeof(a) + sizeof(b);
if (c.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*c, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 0);
}
if (d.has_value()) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  memcpy(buf, &*d, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
  *mask_ptr |= (1 << 1);
}
```

Маски полей: парсинг

Читаем маску

```
uint8_t mask;
if (size < sizeof(mask) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
memcpy(&mask, buf, sizeof(mask));
if ((mask & ~((1 << 0) | (1 << 1)) != 0)
  [[unlikely]] return false;
memcpy(&a, buf + sizeof(mask), sizeof(a));
memcpy(&b, buf + sizeof(mask) + sizeof(a), sizeof(b));
buf += sizeof(mask) + sizeof(a) + sizeof(b);
size -= sizeof(mask) + sizeof(a) + sizeof(b);
if ((mask & (1 << 0)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  c = 0;
  memcpy(&*c, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else c.reset();
if ((mask & (1 << 1)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  d = 0;
  memcpy(&*d, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else d.reset();
```

Маски полей: парсинг

Безусловно присутствующие поля сериализуем как обычно

```
uint8_t mask;
if (size < sizeof(mask) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
memcpy(&mask, buf, sizeof(mask));
if ((mask & ~((1 << 0) | (1 << 1)) != 0)
  [[unlikely]] return false;
memcpy(&a, buf + sizeof(mask), sizeof(a));
memcpy(&b, buf + sizeof(mask) + sizeof(a), sizeof(b));
buf += sizeof(mask) + sizeof(a) + sizeof(b);
size -= sizeof(mask) + sizeof(a) + sizeof(b);
if ((mask & (1 << 0)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  c = 0;
  memcpy(&*c, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else c.reset();
if ((mask & (1 << 1)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  d = 0;
  memcpy(&*d, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else d.reset();
```

Маски полей: парсинг

В зависимости от того, какие биты поставлены у маски, читаем соответствующие поля

Остальные поля инициализируются “пустым” значением

```
uint8_t mask;
if (size < sizeof(mask) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
memcpy(&mask, buf, sizeof(mask));
if ((mask & ~((1 << 0) | (1 << 1)) != 0)
  [[unlikely]] return false;
memcpy(&a, buf + sizeof(mask), sizeof(a));
memcpy(&b, buf + sizeof(mask) + sizeof(a), sizeof(b));
buf += sizeof(mask) + sizeof(a) + sizeof(b);
size -= sizeof(mask) + sizeof(a) + sizeof(b);
if ((mask & (1 << 0)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  c = 0;
  memcpy(&*c, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else c.reset();
if ((mask & (1 << 1)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  d = 0;
  memcpy(&*d, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else d.reset();
```

Маски полей: парсинг

В зависимости от того, какие биты поставлены у маски, читаем соответствующие поля

Остальные поля инициализируются “пустым” значением

```
uint8_t mask;
if (size < sizeof(mask) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
memcpy(&mask, buf, sizeof(mask));
if ((mask & ~((1 << 0) | (1 << 1)) != 0)
  [[unlikely]] return false;
memcpy(&a, buf + sizeof(mask), sizeof(a));
memcpy(&b, buf + sizeof(mask) + sizeof(a), sizeof(b));
buf += sizeof(mask) + sizeof(a) + sizeof(b);
size -= sizeof(mask) + sizeof(a) + sizeof(b);
if ((mask & (1 << 0)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  c = 0;
  memcpy(&*c, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else c.reset();
if ((mask & (1 << 1)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  d = 0;
  memcpy(&*d, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else d.reset();
```

Маски полей: парсинг

Проверяем, что не соответствующие полям биты равны нулю

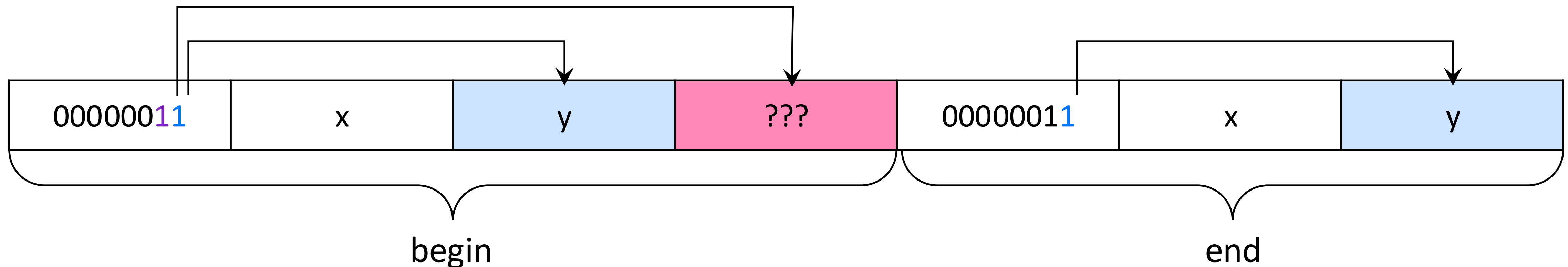
```
uint8_t mask;
if (size < sizeof(mask) + sizeof(a) + sizeof(b))
  [[unlikely]] return false;
memcpy(&mask, buf, sizeof(mask));
if ((mask & ~((1 << 0) | (1 << 1)) != 0)
  [[unlikely]] return false;
memcpy(&a, buf + sizeof(mask), sizeof(a));
memcpy(&b, buf + sizeof(mask) + sizeof(a), sizeof(b));
buf += sizeof(mask) + sizeof(a) + sizeof(b);
size -= sizeof(mask) + sizeof(a) + sizeof(b);
if ((mask & (1 << 0)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  c = 0;
  memcpy(&*c, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else c.reset();
if ((mask & (1 << 1)) != 0) {
  if (size < sizeof(int32_t)) [[unlikely]] return false;
  d = 0;
  memcpy(&*d, buf, sizeof(int32_t));
  buf += sizeof(int32_t);
  size -= sizeof(int32_t);
} else d.reset();
```

Маски полей

- Не соответствующие полям биты должны быть равны нулю

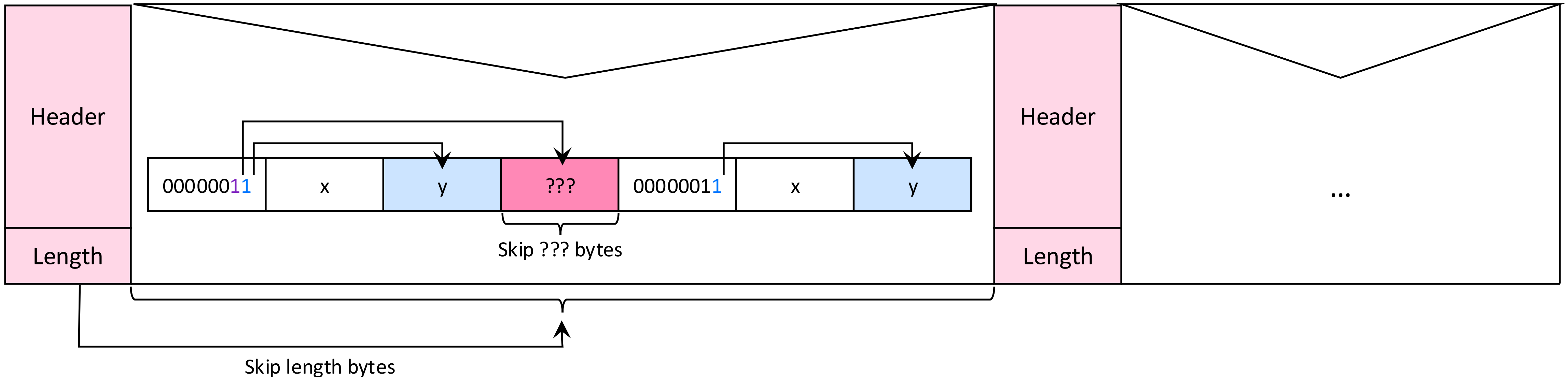
```
point mask: # x: int y: mask.0?int = Point
segment begin: point end: point = Segment
```

- Не можем ни прочитать, **ни пропустить** незнакомое поле
- Не знаем, сколько байт оно занимает
- Не знаем, с какого байта читать следующий объект



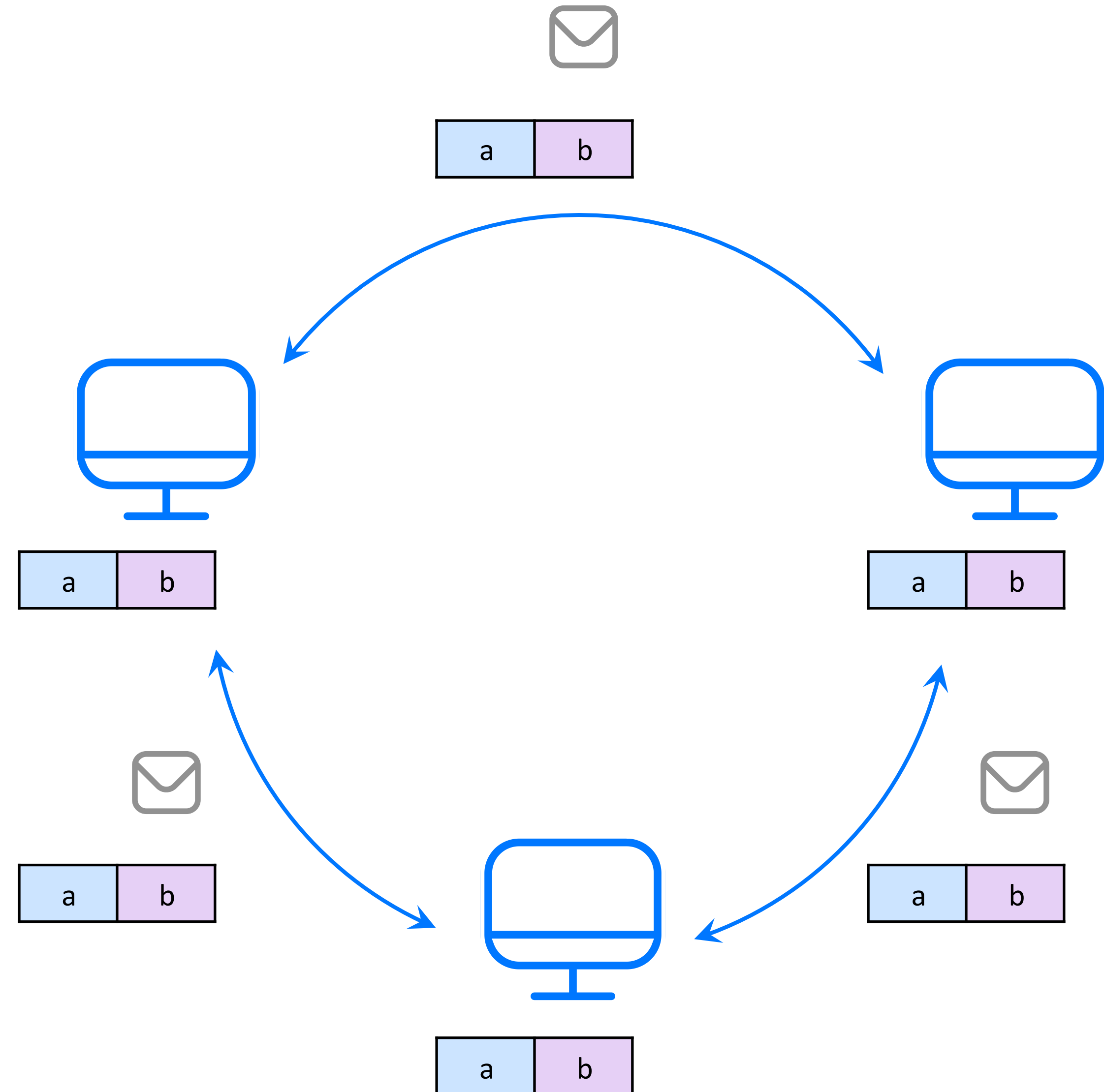
Маски полей

- Перед каждым запросом пишется его длина в байтах
- Если в ходе парсинга запроса встретили тип, который не можем прочитать — пропускаем весь запрос целиком
- Знаем, сколько байт нужно пропустить



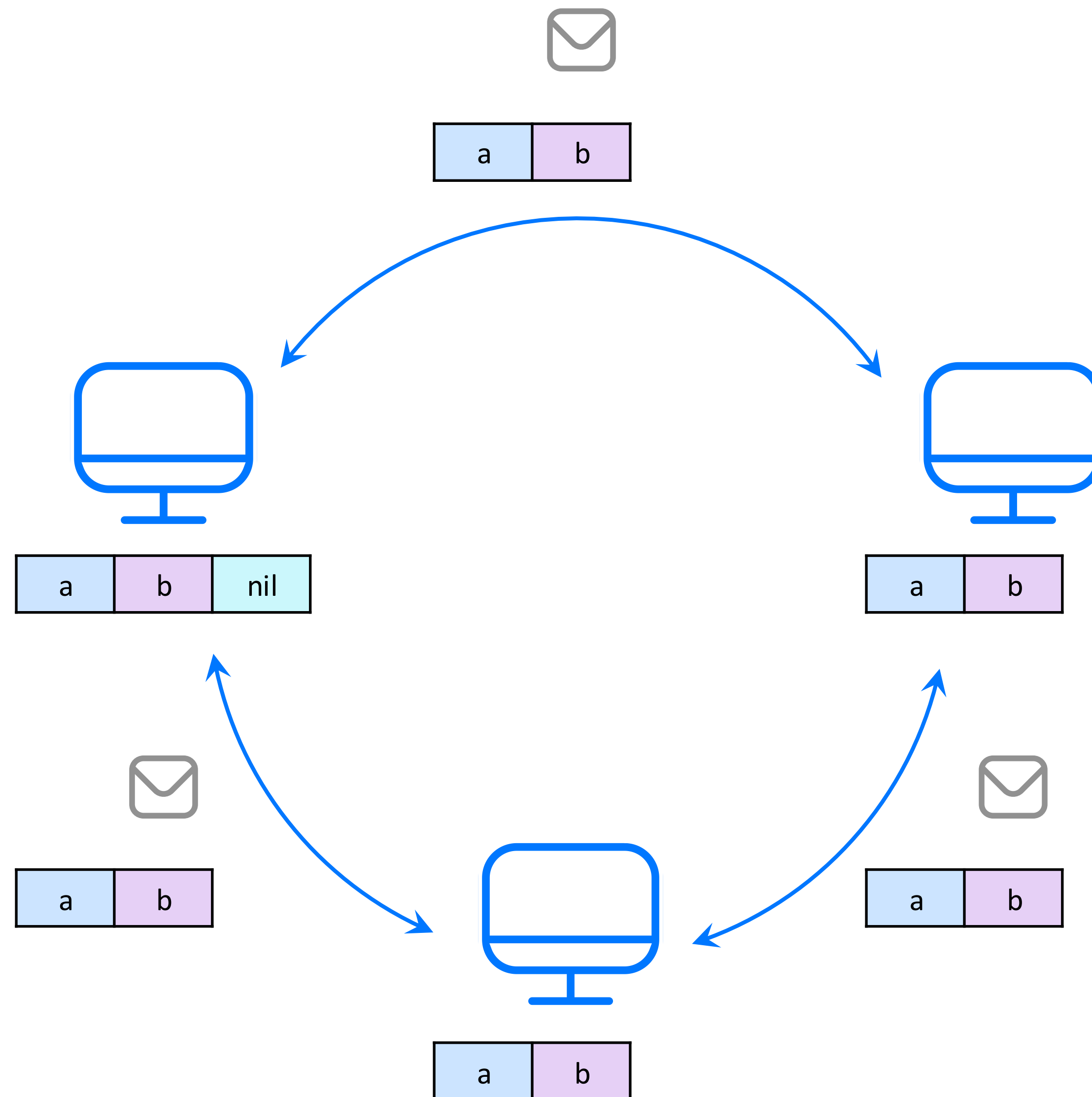
Обновление схемы

- Сообщение от отправителя с новой схемой нельзя отправлять на получателя со старой схемой
- Непонятно, кого обновлять первым



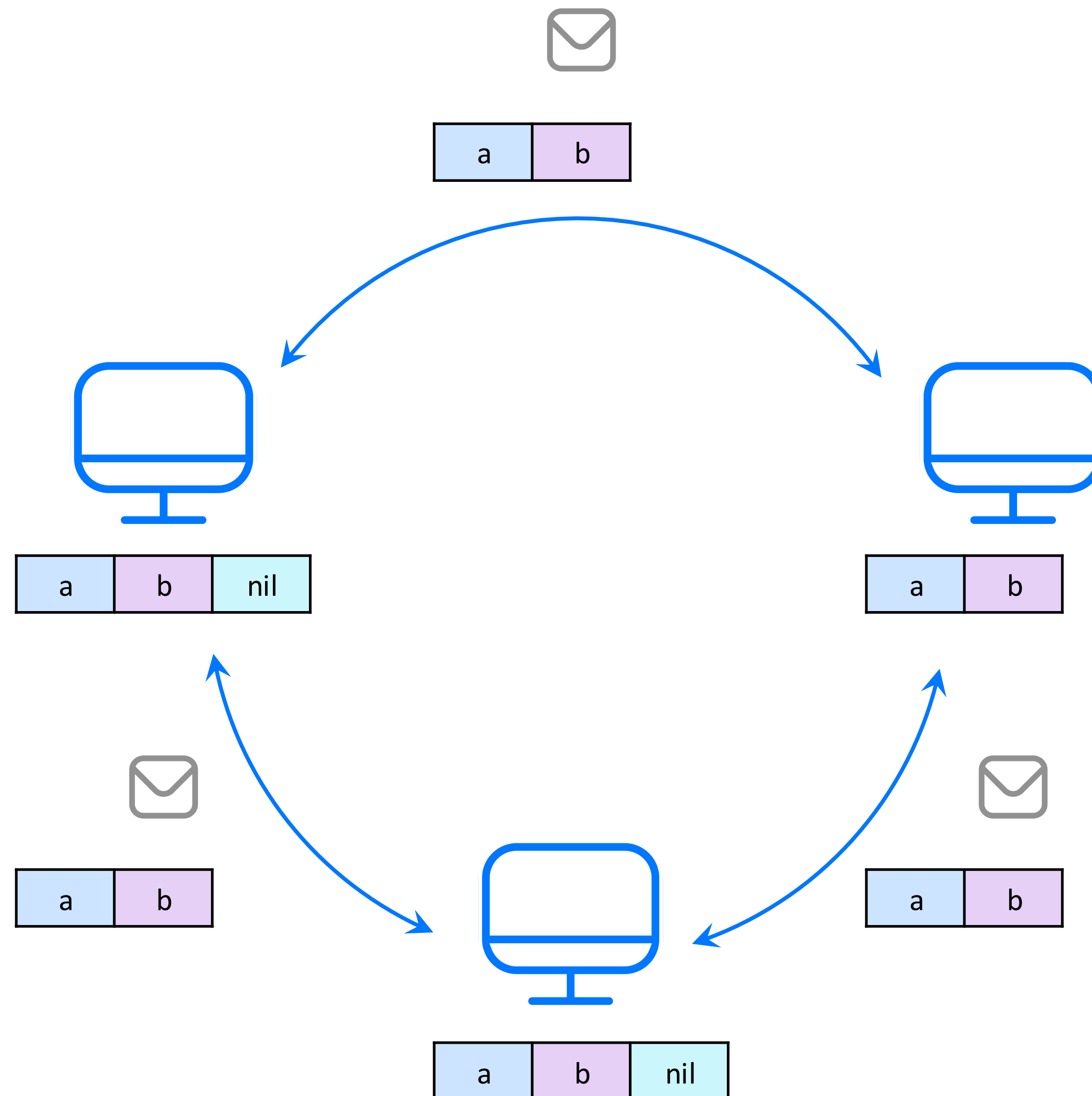
Обновление схемы

- Первым деплоем учим **всех** читать сообщения в новом формате
- Отправляются сообщения в старом формате
- У нового поля значение по умолчанию
- Обновлять можно **по одному в любом порядке**



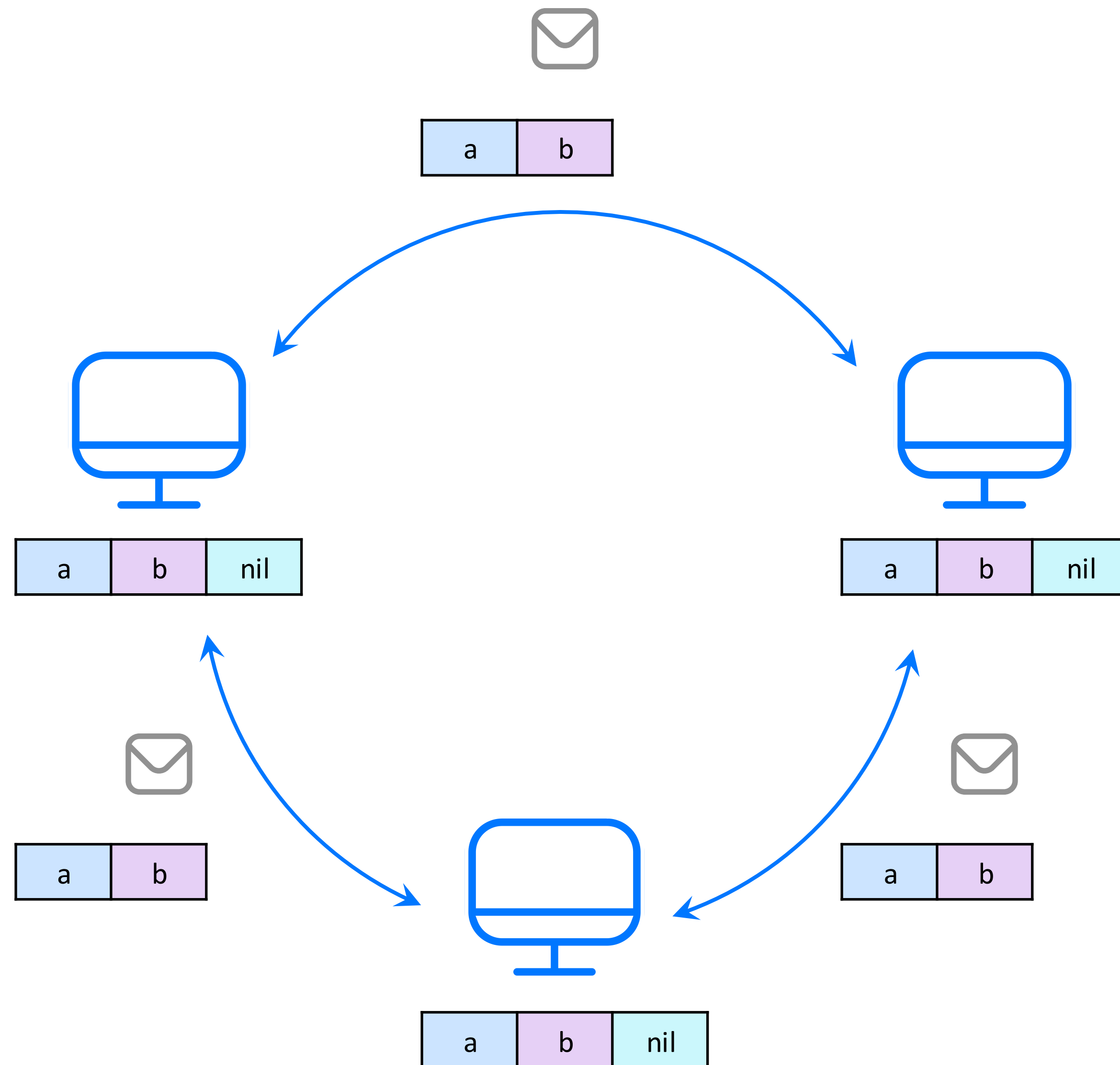
Обновление схемы

- Первым деплоем учим **всех** читать сообщения в новом формате
- Отправляются сообщения в старом формате
- У нового поля значение по умолчанию
- Обновлять можно **по одному в любом порядке**



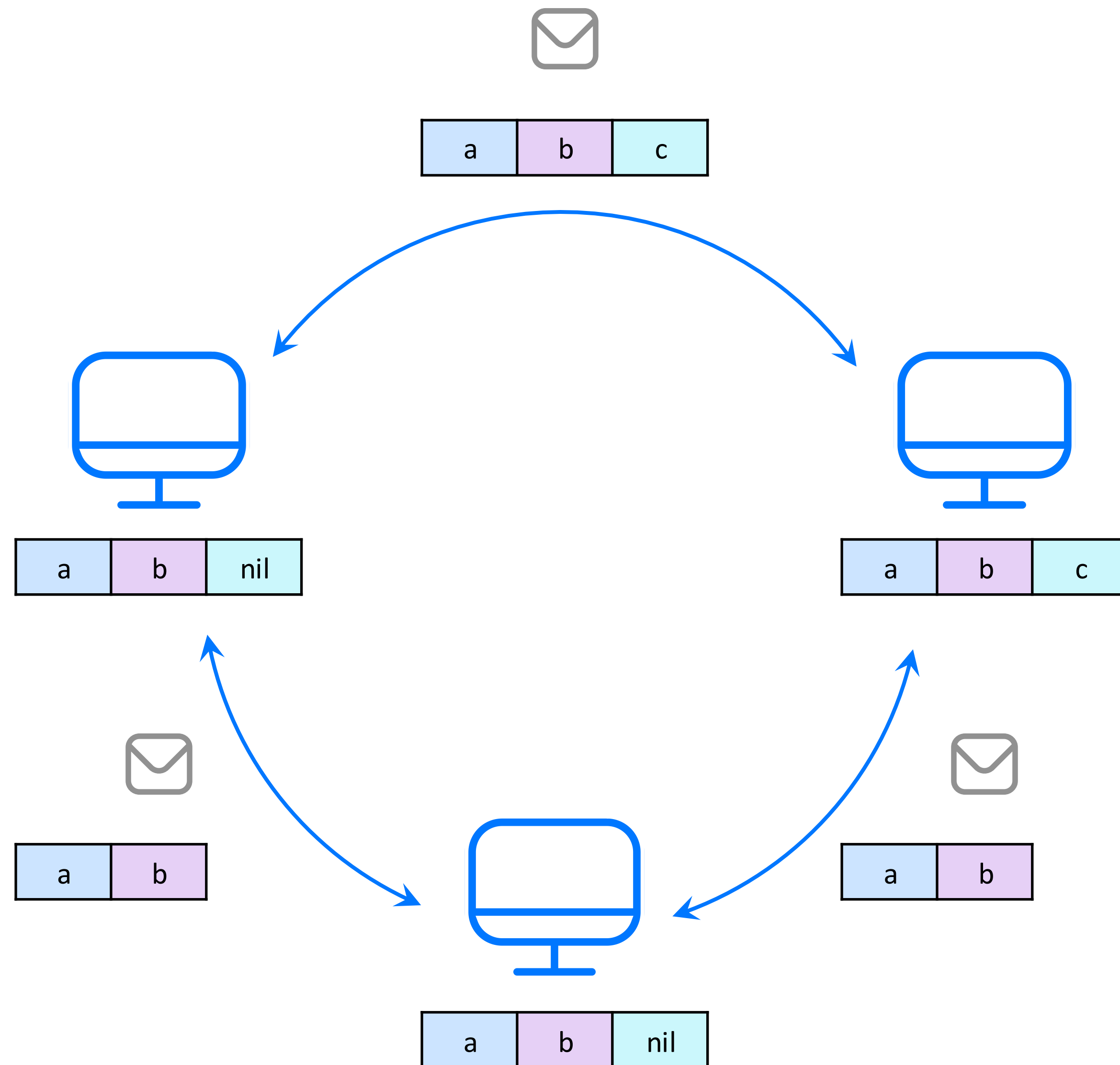
Обновление схемы

- Научив всех читать сообщения в новом
  формате, можем переходить к следующей фазе
- Сервера после второго обновления начинают отправлять сообщения
  с новым полем



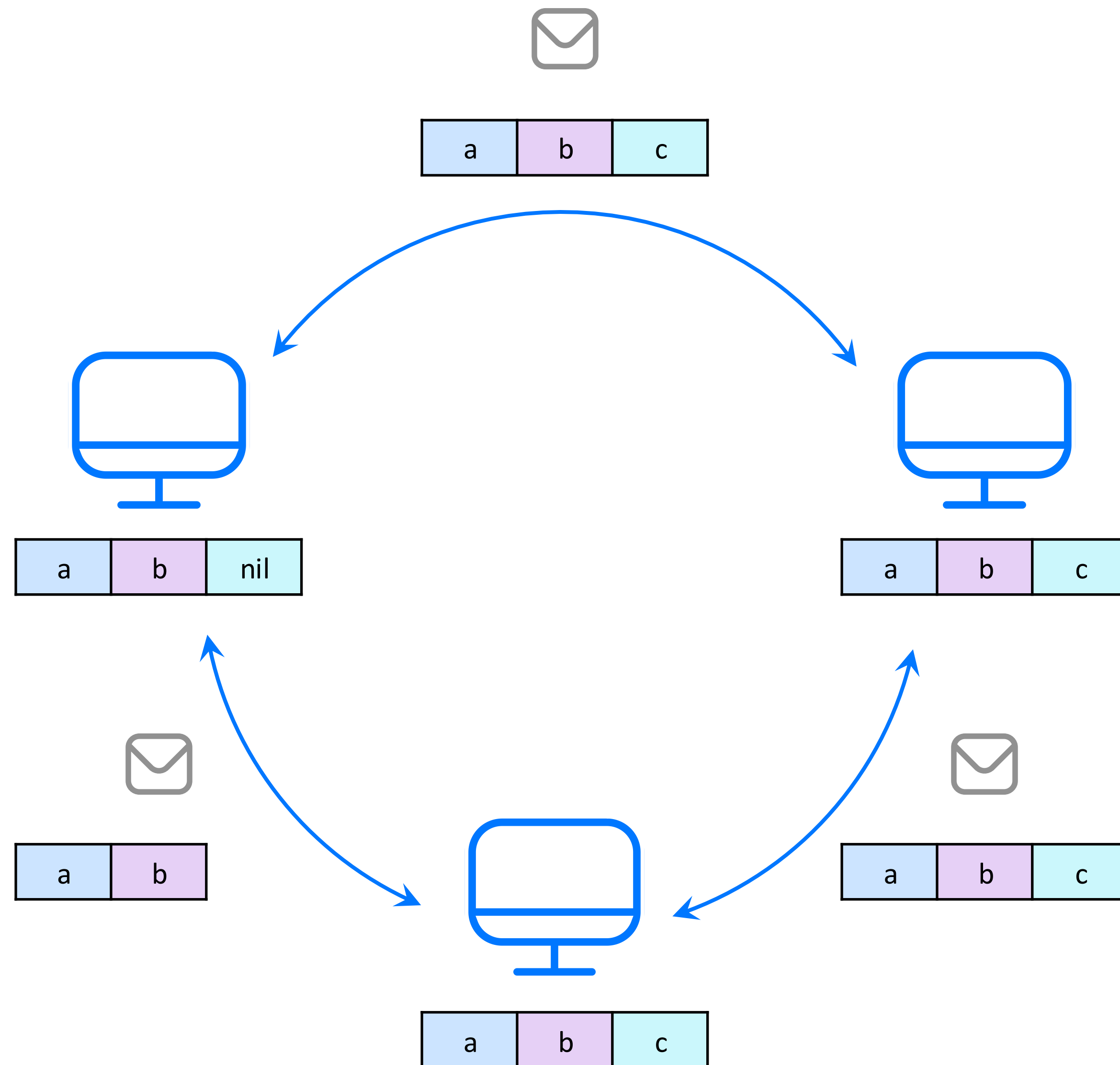
Обновление схемы

- Научив всех читать сообщения в новом
  формате, можем переходить к следующей фазе
- Сервера после второго обновления начинают отправлять сообщения
  с новым полем



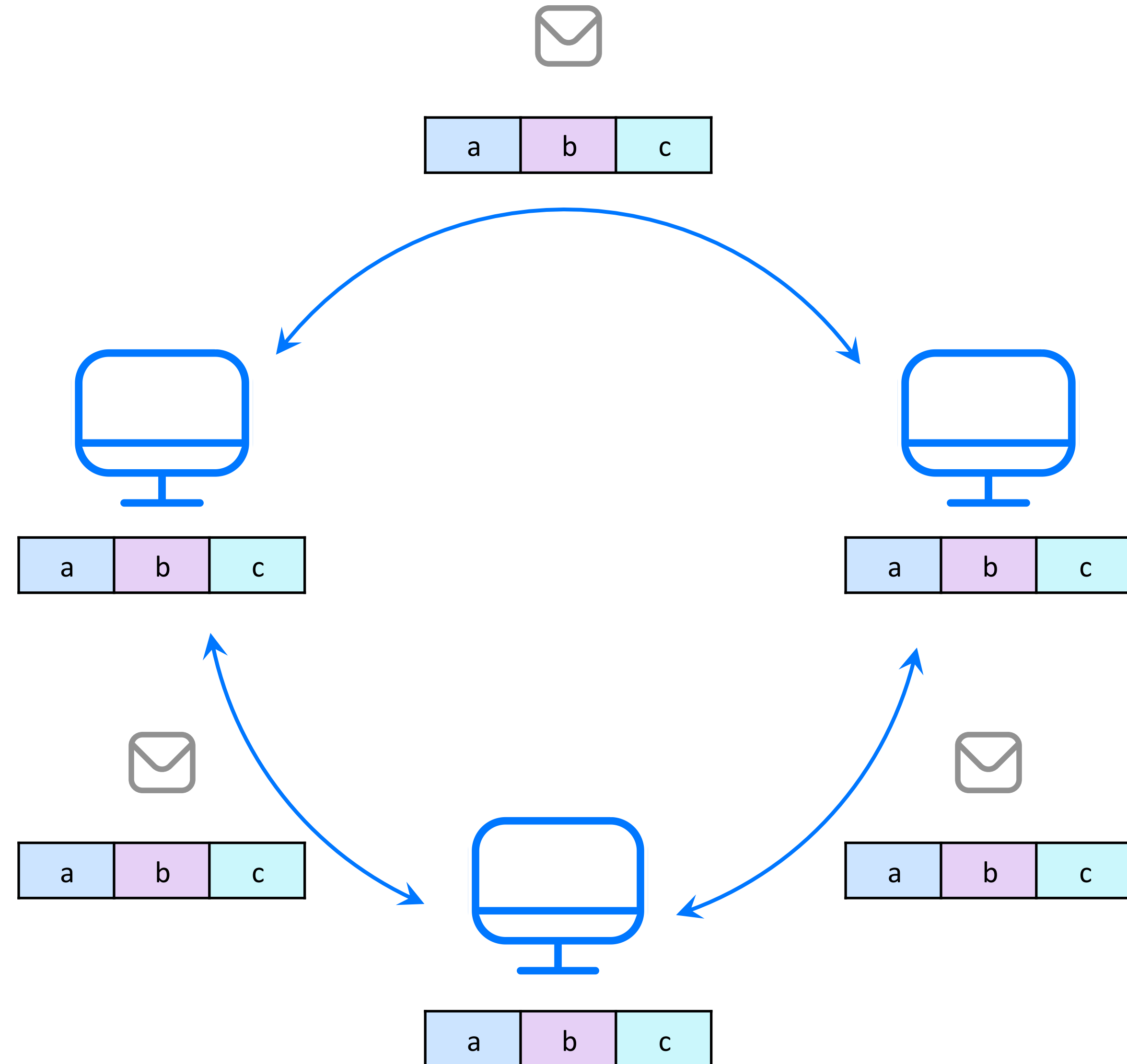
Обновление схемы

- Сервера после второго обновления начинают отправлять сообщения с новым полем
- Включать отправку нового поля можно в любом порядке
- Читать новое поле умеют все



Обновление схемы

- Теперь обновление схемы можно считать успешным
- Все сообщения отправляются и принимаются в новом формате



Возможность обновления схемы

- В схему с маской можно добавить новое поле

```
data mask: # x: int y: mask.0?long = Data;
```

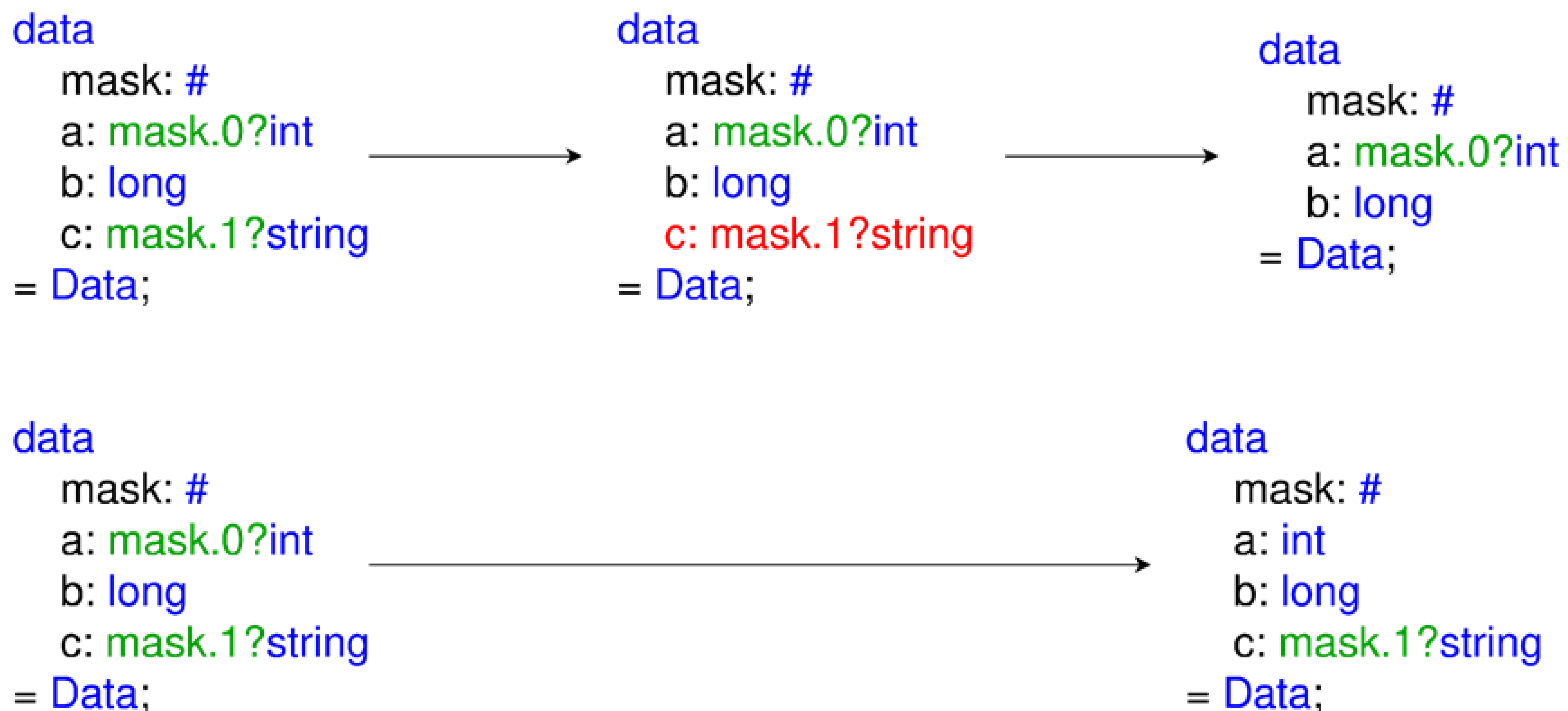
- В схему без маски нельзя

```
data x: int y: int c: int = Data;
```

With this character's death, the thread of prophecy is severed. Restore a saved game to restore the weave of fate, or persist in the doomed world you have created.

Обновления схемы

В ходе эволюции схемы можно освобождать биты маски для дальнейшего использования



Поддержка неограниченного числа полей

- Не можем добавлять новые поля потому что у маски закончились биты

```
data
    mask: #
    field0: mask.0?int
    field1: mask1?long
    /* …*/
    field6: mask.6?string

    field7: mask7?float
    field8: mask.8?double
= Data;
```

Existing fields

New fields

Поддержка неограниченного числа полей

- `mask2` сериализуется если старший бит `mask` равен единице

```
data
    mask: #
    field0: mask.0?int
    field1: mask1?long
    /* …*/
    field6: mask.6?string
    mask2: mask.7?#
= Data;
```

Поддержка неограниченного числа полей

- Маска может зависеть от другой маски
- Теперь у нас есть ещё биты для условной сериализации других полей

```
data
    mask: #
    field0: mask.0?int
    field1: mask1?long
    /* …*/
    field6: mask.6?string
    mask2: mask.7?#
    field7: mask2.0?float
    field8: mask.2.1?double
= Data;
```

Бенчмарки работы с масками полей

Пять случайных полей непустые

```
missing
  mask: #
  a:     mask.0?int
  b:     mask.1?int
  c:     mask.2?int
  d:     mask.3?int
  e:     mask.4?int
  f:     mask.5?int
  g:     mask.6?int
  h:     mask.7?int
  i:     mask.8?int
  j:     mask.9?int
= Missing;
```

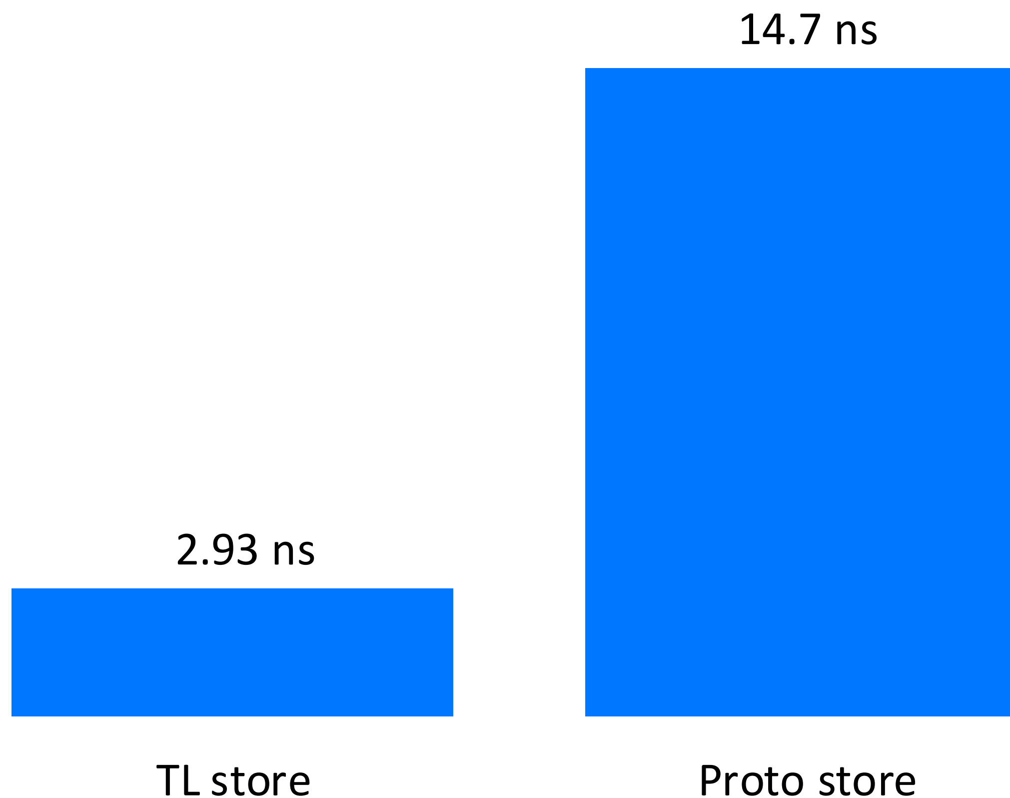
```
static void BM_TL_missing_store(benchmark::State& state) {
    tl_missing_t from{prepare_tl()};
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = from.tl_store(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_missing_store(benchmark::State& state) {
    protobench::Missing from{prepare_proto()};
    std::array<uint8_t, 4096> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        from.SerializeToArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}
```

```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
  int32 f = 6;
  int32 g = 7;
  int32 h = 8;
  int32 i = 9;
  int32 j = 10;
}
```

Бенчмарки работы с масками полей

Пять случайных полей непустые

```
missing
  mask: #
  a:     mask.0?int
  b:     mask.1?int
  c:     mask.2?int
  d:     mask.3?int
  e:     mask.4?int
  f:     mask.5?int
  g:     mask.6?int
  h:     mask.7?int
  i:     mask.8?int
  j:     mask.9?int
= Missing;
```




```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
  int32 f = 6;
  int32 g = 7;
  int32 h = 8;
  int32 i = 9;
  int32 j = 10;
}
```

Бенчмарки работы с масками полей

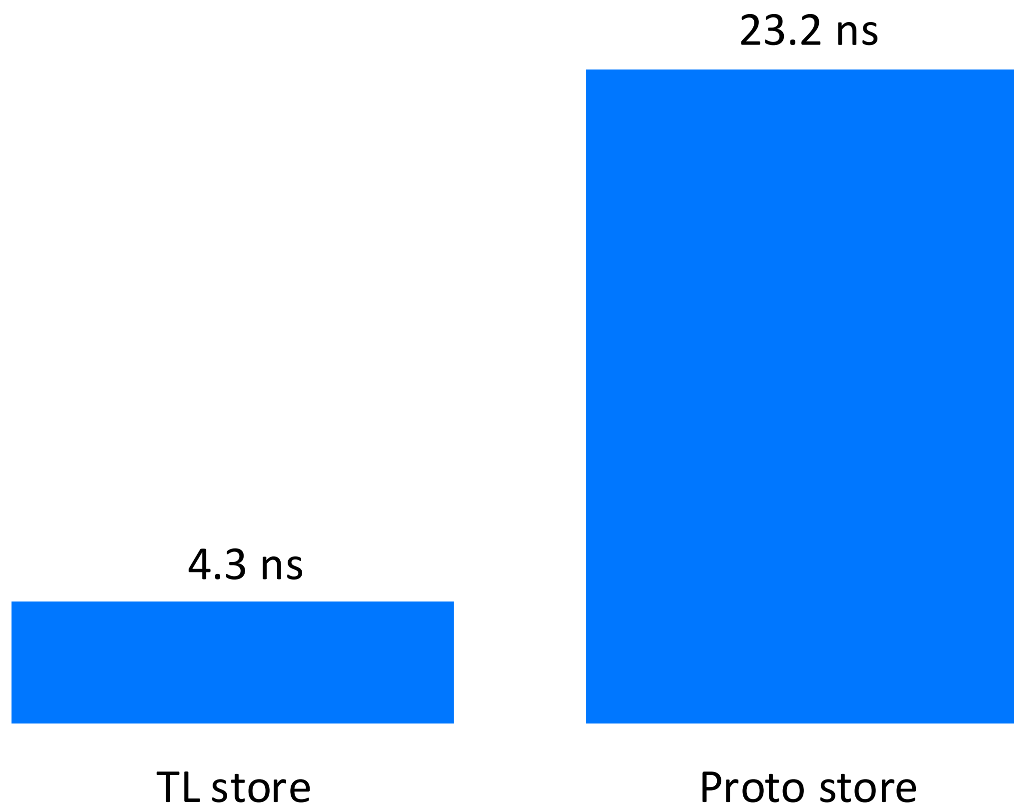
```
missing
  mask: #
  a:    mask.0?int
  b:    mask.1?int
  c:    mask.2?int
  d:    mask.3?int
  e:    mask.4?int
  f:    mask.5?int
  g:    mask.6?int
  h:    mask.7?int
  i:    mask.8?int
  j:    mask.9?int
= Missing;
```

```
static void BM_TL_missing_parse(benchmark::State& state) {
    tl_missing_t from{prepare_tl()};
    std::array<uint8_t, 4096> buf{};
    std::ignore = from.tl_store(buf.data(), buf.size());
    tl_missing_t to{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = to.tl_fetch(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_missing_parse(benchmark::State& state) {
    protobench::Missing from{prepare_proto()};
    std::array<uint8_t, 4096> buf{};
    from.SerializeToArray(buf.data(), buf.size());
    protobench::Missing to{};
    for ([[maybe_unused]] const auto _ : state) {
        to.ParseFromArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}
```

```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
  int32 f = 6;
  int32 g = 7;
  int32 h = 8;
  int32 i = 9;
  int32 j = 10;
}
```

Бенчмарки работы с масками полей

```
missing
  mask: #
  a:     mask.0?int
  b:     mask.1?int
  c:     mask.2?int
  d:     mask.3?int
  e:     mask.4?int
  f:     mask.5?int
  g:     mask.6?int
  h:     mask.7?int
  i:     mask.8?int
  j:     mask.9?int
= Missing;
```

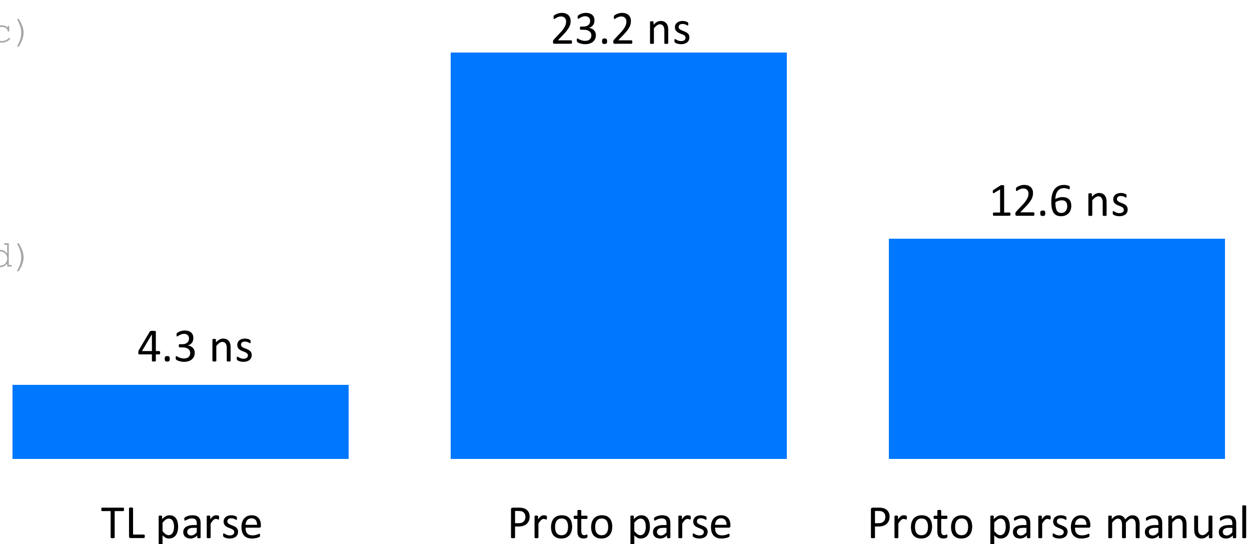



```
message Simple {
  int32 a = 1;
  int32 b = 2;
  int32 c = 3;
  int32 d = 4;
  int32 e = 5;
  int32 f = 6;
  int32 g = 7;
  int32 h = 8;
  int32 i = 9;
  int32 j = 10;
}
```

Бенчмарки работы с масками полей

```
auto tag = input.ReadTagNoLastTag();
if (tag == 8u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &a)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 16u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &b)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 24u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &c)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 32u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &d)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
/* parse e, f, g, h, I, j fields */
```

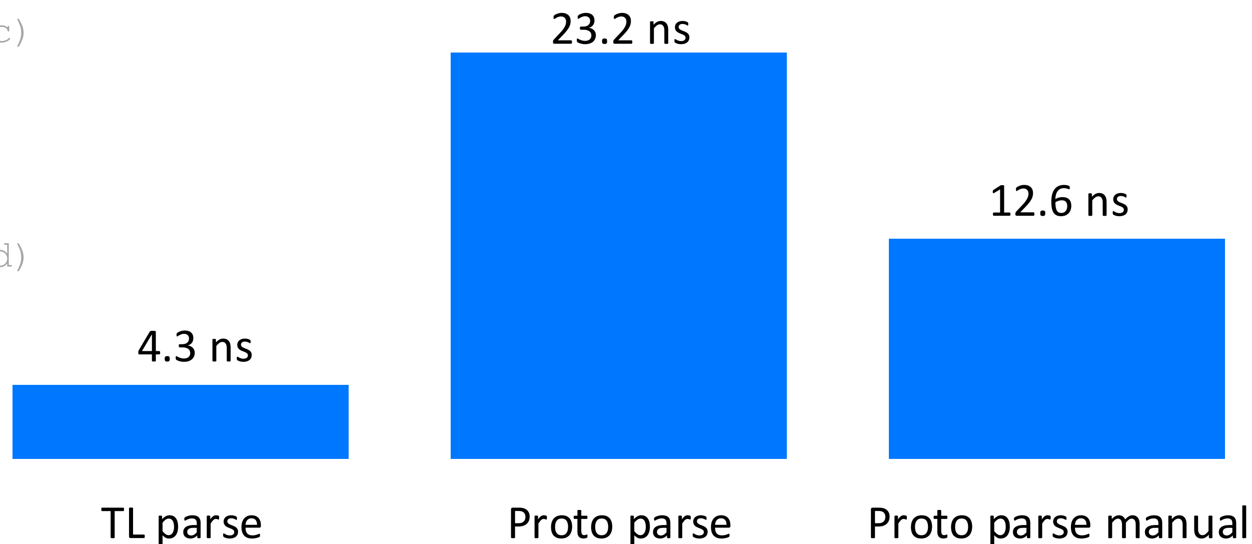
- Смотрим на текущий id поля
- Если он равен ожидаемому, парсим значение поля, читаем следующий id поля
- В противном случае переходим к следующему полю



Бенчмарки работы с масками полей

```
auto tag = input.ReadTagNoLastTag();
if (tag == 8u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &a)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 16u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &b)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 24u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &c)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 32u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &d)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
/* parse e, f, g, h, I, j fields */
```

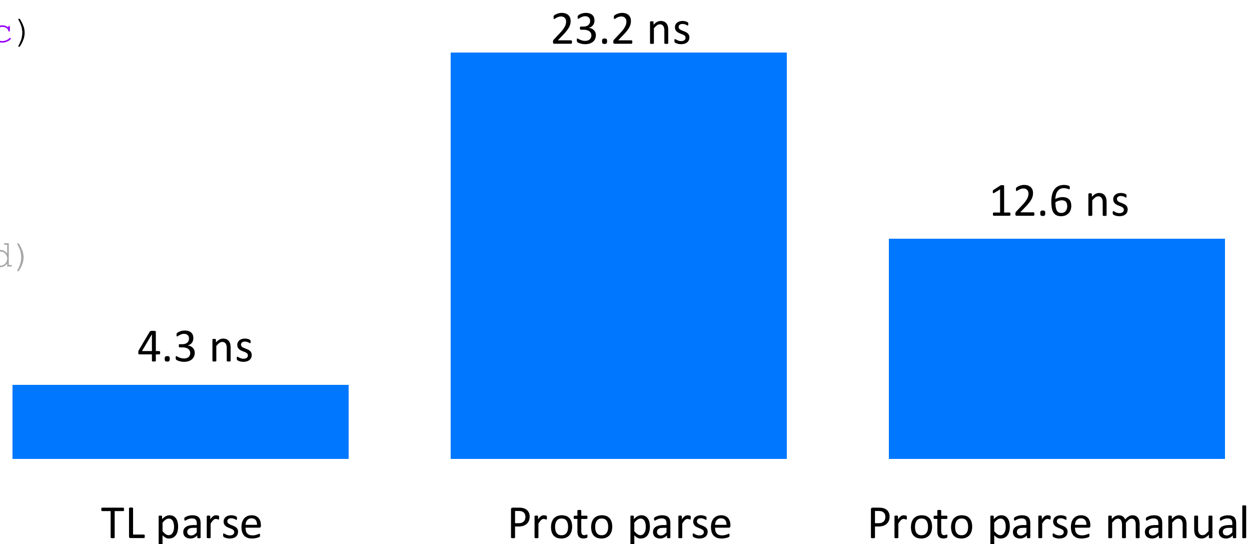
- Смотрим на текущий id поля
- Если он равен ожидаемому, парсим значение поля, читаем следующий id поля
- В противном случае переходим к следующему полю



Бенчмарки работы с масками полей

```
auto tag = input.ReadTagNoLastTag();
if (tag == 8u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &a)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 16u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &b)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 24u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &c)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 32u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &d)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
/* parse e, f, g, h, I, j fields */
```

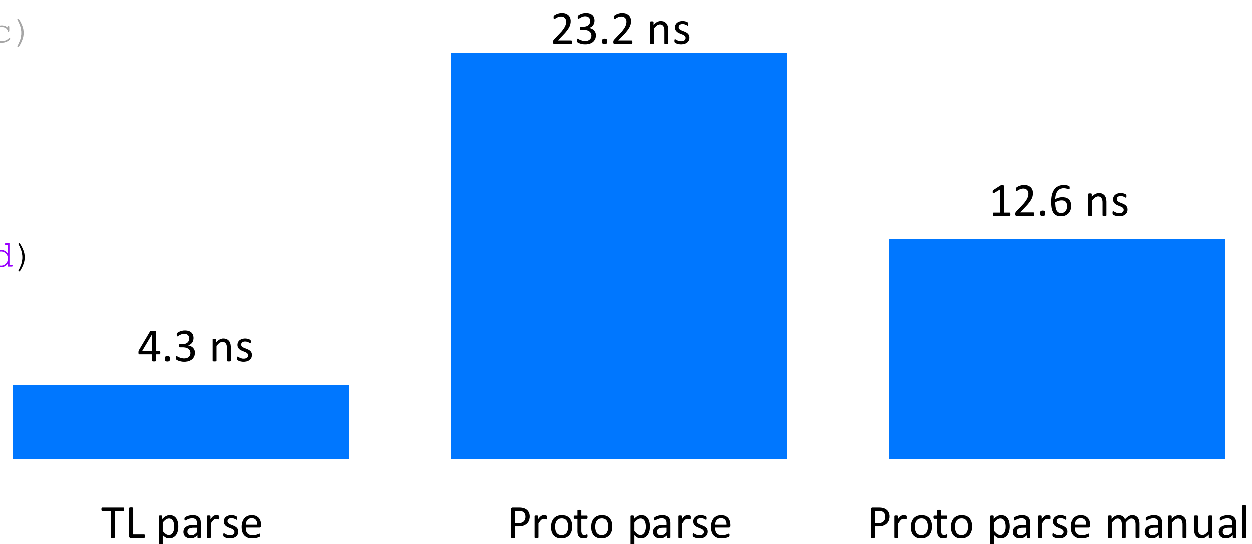
- Смотрим на текущий id поля
- Если он равен ожидаемому, парсим значение поля, читаем следующий id поля
- В противном случае переходим к следующему полю



Бенчмарки работы с масками полей

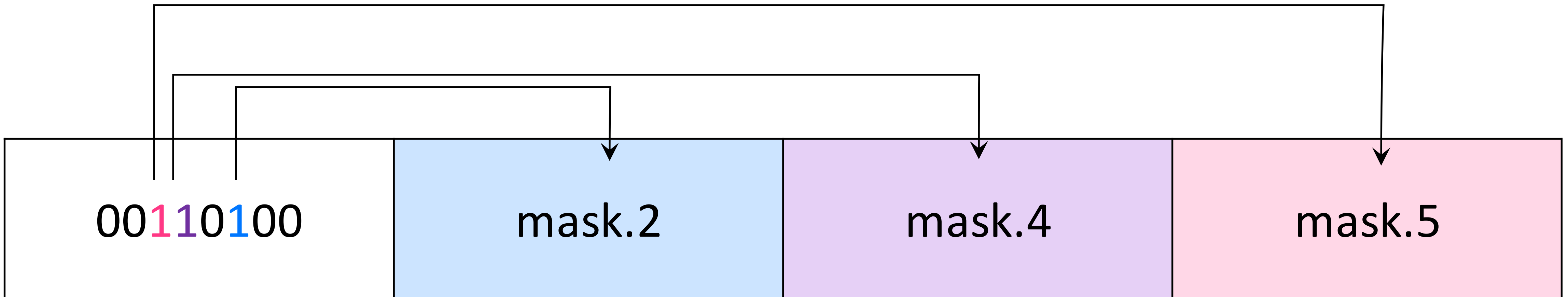
```
auto tag = input.ReadTagNoLastTag();
if (tag == 8u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &a)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 16u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &b)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 24u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &c)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
if (tag == 32u) {
  if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &d)
    [[unlikely]] std::abort();
  tag = input.ReadTagNoLastTag();
}
/* parse e, f, g, h, I, j fields */
```

- Смотрим на текущий id поля
- Если он равен ожидаемому, парсим значение поля, читаем следующий id поля
- В противном случае переходим к следующему полю



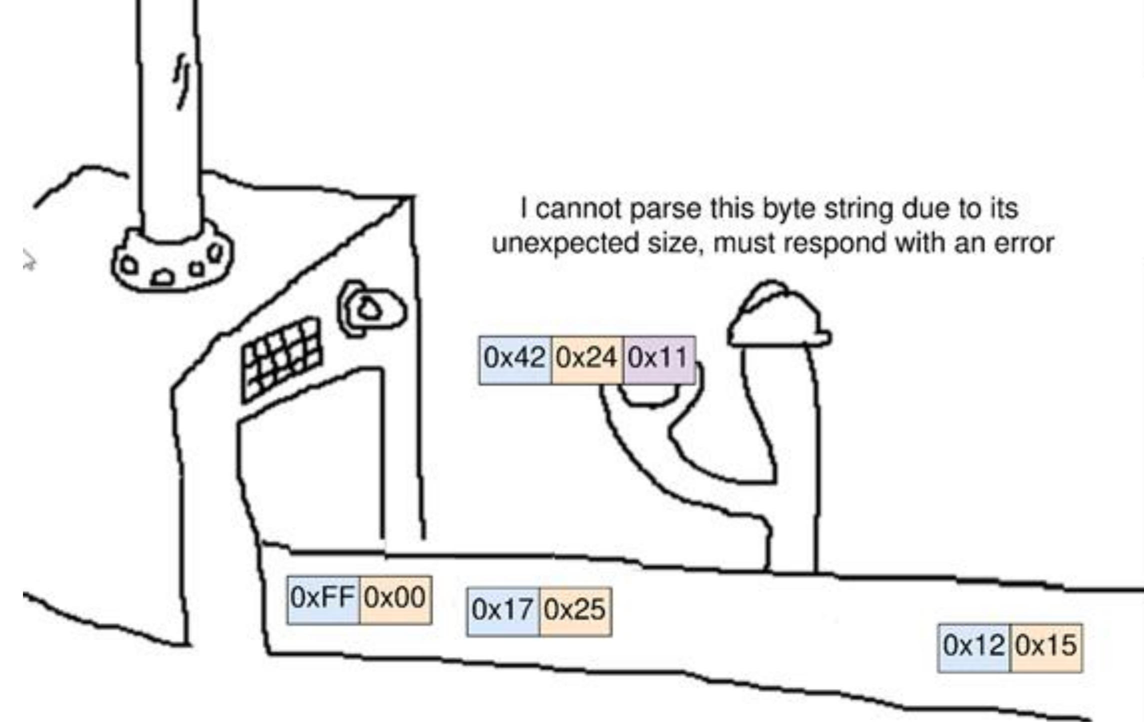
Бенчмарки работы с масками полей

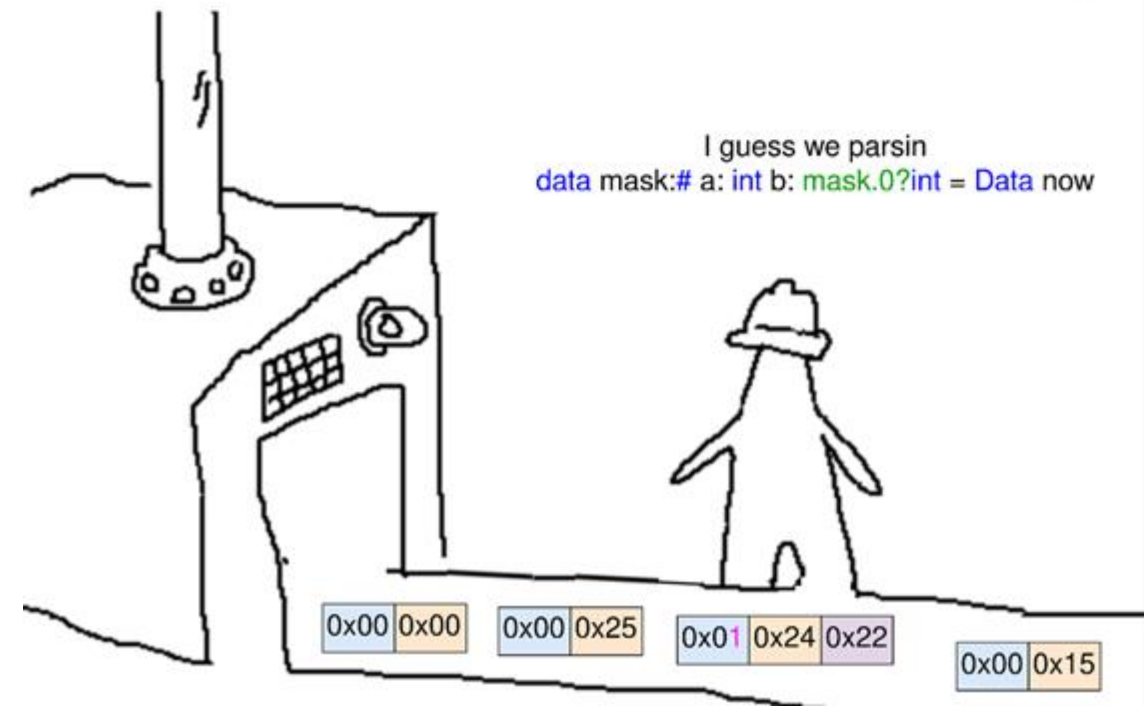
- Чтение одной маски в TL позволяет сделать вывод о существовании сразу нескольких полей
- Protobuf обязан делать этот вывод для каждого поля в отдельности



Использование масок полей

- Маски полей — достаточно мощный инструмент для обеспечения **backward compatibility**
- Но не подходят для **forward compatibility**
- Работают быстро
- Приходится использовать с умом
- Жертвуем удобством эволюции схем в пользу скорости

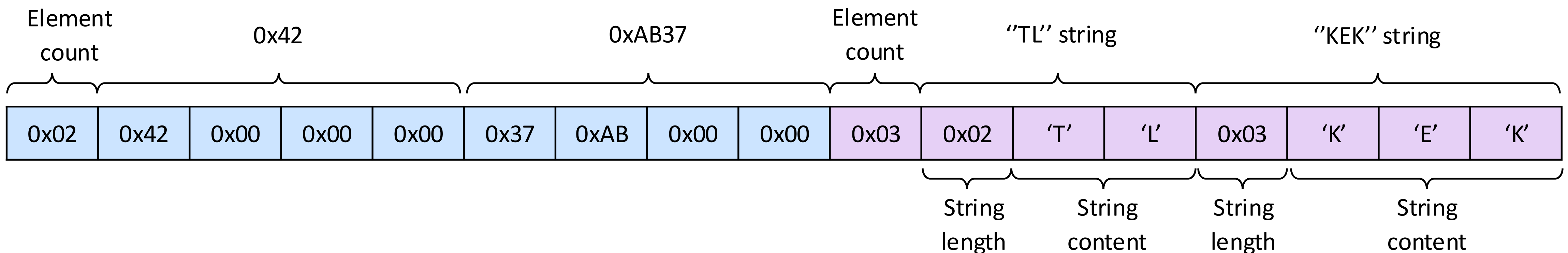




Структура TL: массивы

```
vectors
   ints: vector int
   strings: vector string
= Vectors;
```

- При сериализации массива пишется сначала количество элементов, потом его элементы
- Разумеется, поддерживаются массивы нетривиальных типов (массивы массивов целых чисел, массивы строк, etc)



Бенчмарки работы с массивами

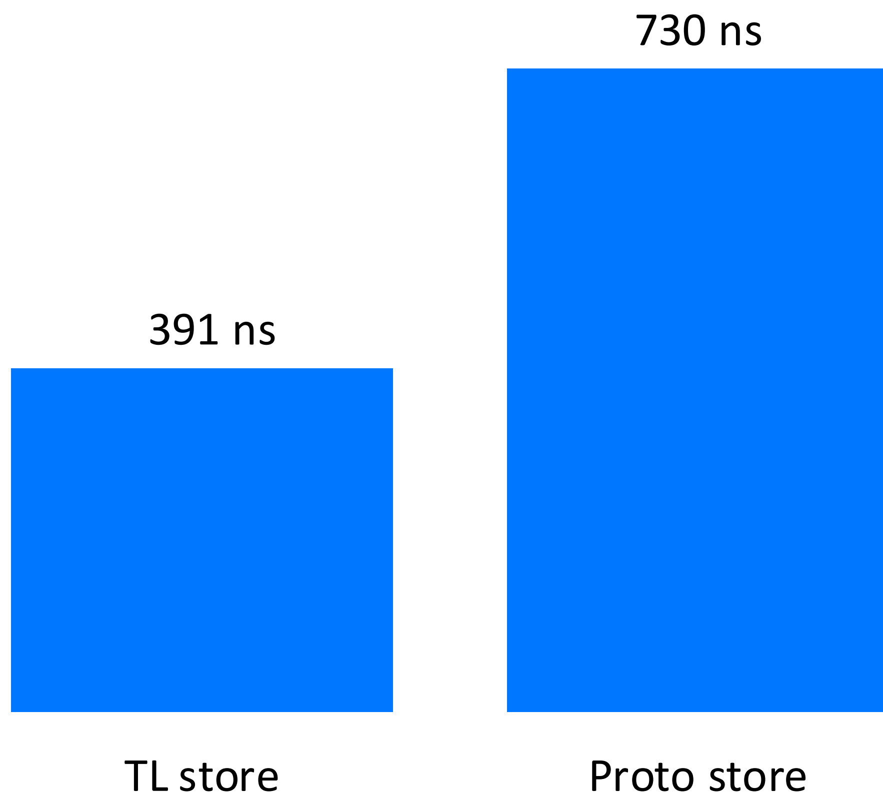
100 строк, 80 коротких и 20 длинных

```
static void BM_TL_array_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    tl_array_t from{};
    for (uint32_t i = 0; i < 100; ++i)
        from.lines.emplace_back((i % 5 == 0) ? 1000 : 10, dist(rng));
    std::array<uint8_t, 1024 * 1024> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = from.tl_store(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_array_store(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    protobench::Array from{};
    for (uint32_t i = 0; i < 100; ++i)
        from.mutable_lines()->Add(std::string((i % 5 == 0) ? 1000 : 10, dist(rng)));
    std::array<uint8_t, 1024 * 1024> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        from.SerializeToArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}
```

```
array
  lines: vector string
= Array;

message Array {
  repeated bytes lines = 1;
}
```

Бенчмарки работы с массивами

100 строк, 80 коротких и 20 длинных



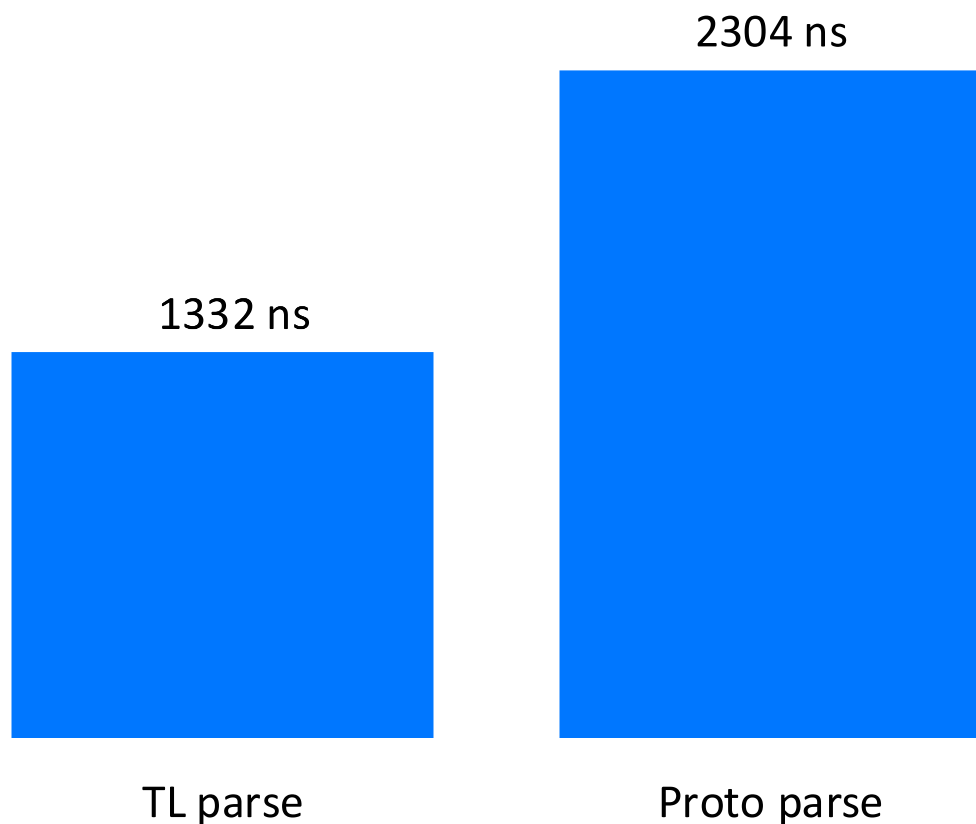

```
array
  lines: vector string
= Array;

message Array {
  repeated bytes lines = 1;
}
```

Бенчмарки работы с массивами

```
static void BM_TL_array_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    tl_array_t from{};
    for (uint32_t i = 0; i < 100; ++i) from.lines.emplace_back((i % 5 == 0) ? 1000 : 10, dist(rng));
    std::array<uint8_t, 1024 * 1024> buf{};
    std::ignore = from.tl_store(buf.data(), buf.size());
    tl_array_t to{};
    to.lines.reserve(128);
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = to.tl_fetch(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_array_parse(benchmark::State& state) {
    std::mt19937 rng{42};
    std::uniform_int_distribution<char> dist{'a', 'z'};
    protobench::Array from{};
    for (uint32_t i = 0; i < 100; ++i) from.mutable_lines()->Add(std::string((i % 5 == 0) ? 1000 : 10, dist(rng)));
    std::array<uint8_t, 1024 * 1024> buf{};
    from.SerializeToArray(buf.data(), buf.size());
    protobench::Array to{};
    to.mutable_lines()->Reserve(128);
    for ([[maybe_unused]] const auto _ : state) {
        to.ParseFromArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}
```

```
array
   lines: vector string
= Array;

message Array {
   repeated bytes lines = 1;
}
```

Бенчмарки работы с массивами



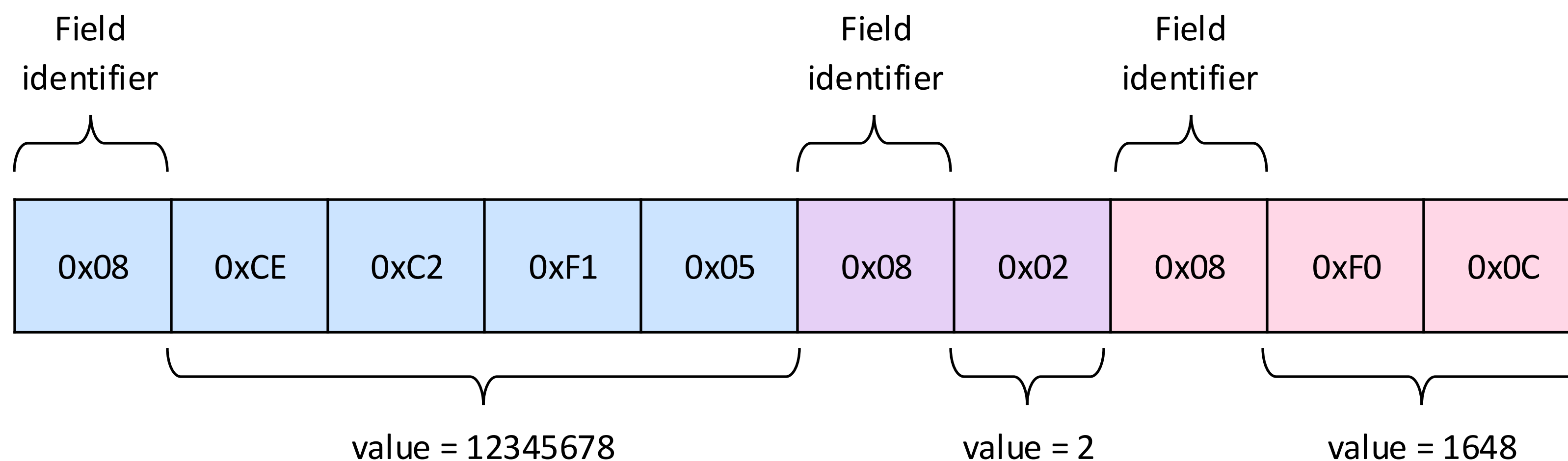

```
array
  lines: vector string
= Array;

message Array {
  repeated bytes lines = 1;
}
```

Массивы в protobuf

- Несколько раз сериализовать пару

`<field id, field value>`

```
message Array {
  repeated int32 values  = 1 [packed = false];
}
```

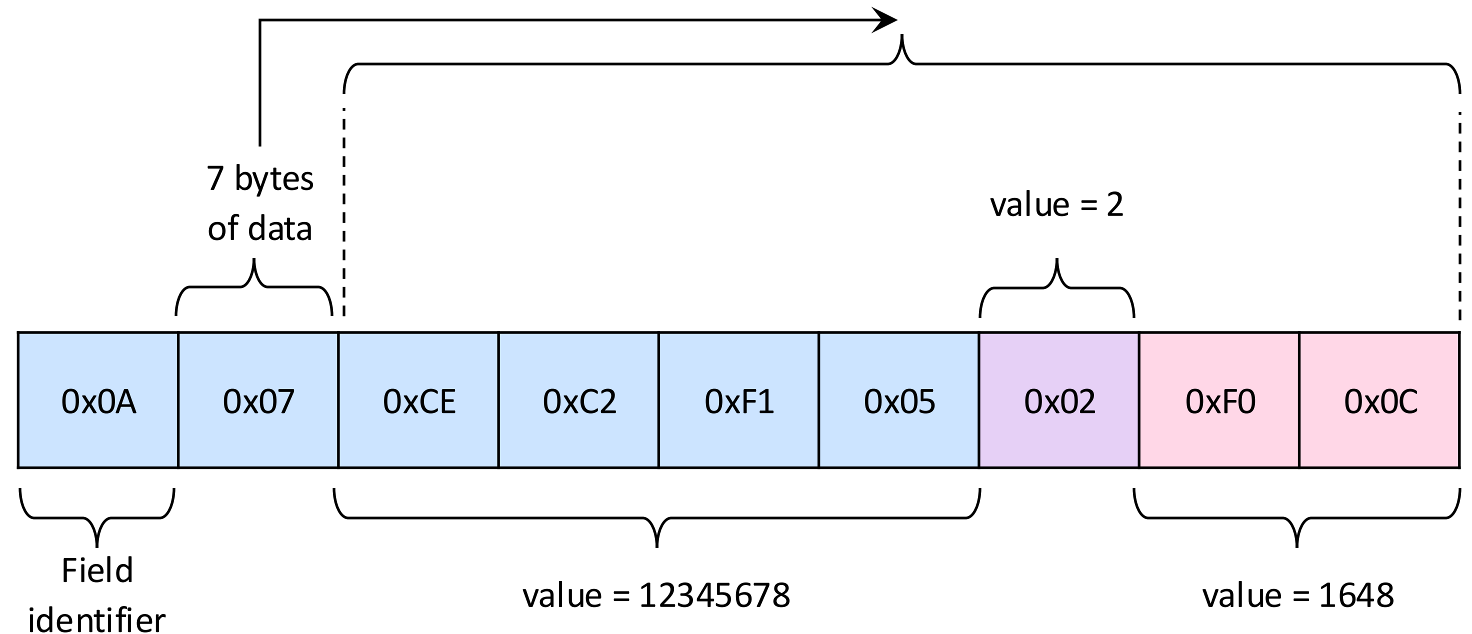


Массивы в protobuf

```
message Array {
  repeated int32 values  = 1 [packed = true];
}
```

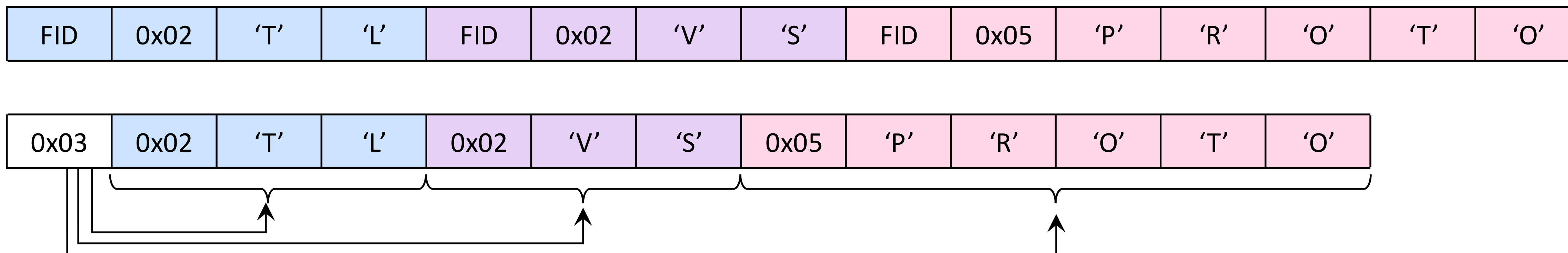
- Идентификатор поля с меткой формата
- Количество байт, занятых элементами массива
- Сами элементы массива

- Только для массивов примитивных типов



Массивы в TL и protobuf

В TL заранее знаем, сколько элементов массива надо прочитать



Сильно упрощает

код парсинга

```
int32_t array_size;
if (size < sizeof(array_size))[[unlikely]] return false;
memcpy(&array_size, buf, 4);
buf += sizeof(array_size);
size -= sizeof(array_size);
lines.resize(array_size);
for (auto& line: lines)
  if (!line.tl_fetch(buf, size)) [[unlikely]] return false;
```

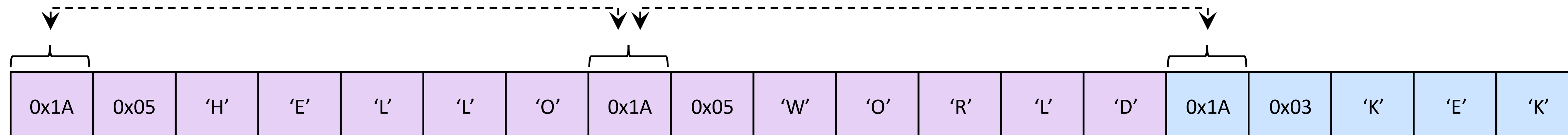
Достоинства массивов в Protobuf

Protobuf позволяет в ходе эволюции схемы удалять/добавлять repeated произвольным образом

```
message Arrays {
    repeated int32 as = 1 [packed = false];
    repeated int32 bc = 2 [packed = true];
    repeated string cs = 3;
    string d = 4;
}
```

```
message Arrays {
    repeated int32 as = 1 [packed = true];
    repeated int32 bs = 2 [packed = false];
    string c = 3;
    repeated string ds = 4;
}
```

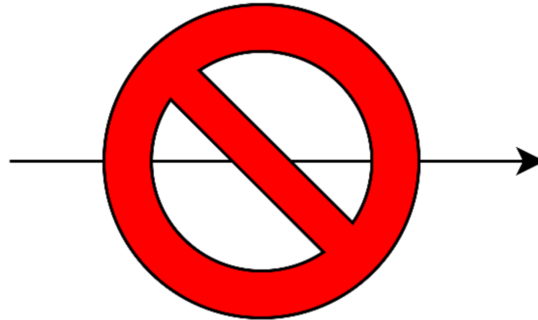
При парсинге сделает самую разумную вещь из возможных



Недостатки массивов в TL

В TL так делать нельзя

```
data
    x: string
    ys: vector string
= Data;
```



```
data
    xs: vector string
    y: string
= Data;
```

0x02	0x05	‘H’	‘E’	‘L’	‘L’	‘O’	0x02	‘T’	‘L’

0x05	‘H’	‘E’	‘L’	‘L’	‘O’

Не можем по бинарному представлению понять способ парсинга поля

Алгебраические типы данных

Геометрическая фигура является или кругом, или прямоугольником

У круга и прямоугольника разный набор свойств-полей

- Радиус у круга
- Две стороны у прямоугольника

Фигура это **тип-сумма**

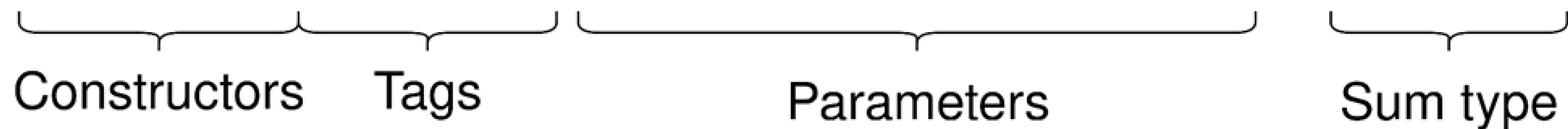
- Tagged union

```
struct circle_t {
    int32_t radius;
};

struct rectangle_t {
    int32_t width;
    int32_t height;
};

std::variant<circle_t, rectangle_t> figure;
```

```
data Figure = Circle Int |              -- Radius
              Rectangle Int Int; -- Width & Height
```

Алгебраические типы данных: описание

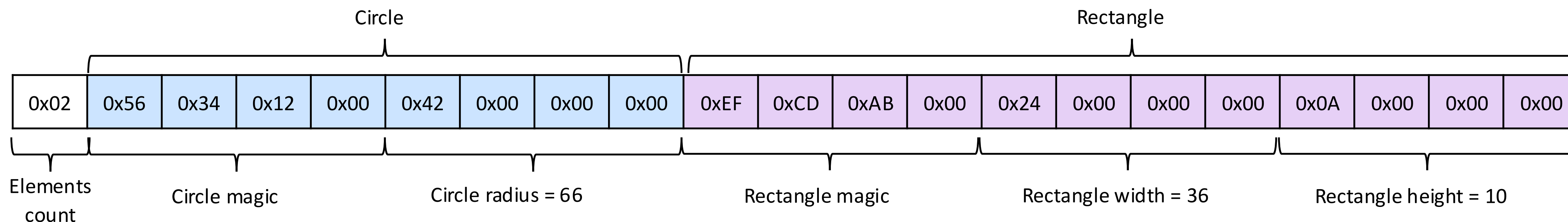
- У алгебраического типа данных есть от одного до бесконечности конструкторов
- У каждого конструктора есть от нуля до бесконечности параметров
  - Числа, строки, структуры, массивы, другие ADT…
- И уникальный тег



Алгебраические типы данных: сериализация

- В начале пишется тег
- По нему понимаем при парсинге, какую из альтернатив читать
- После тега пишутся параметры без дополнительной метаинформации

```
circle#00123456
  radius: int
= Figure;

rectangle#00abcdef
  width: int
  height: int
= Figure;

figures
  figures: vector Figure
= Figures;
```



Алгебраические типы данных: запись

- Смотрим на ту, какую из альтернатив сериализуем

```
struct visitor_t {
  bool operator()(const tl_circle_t& cur_circle) noexcept {
    if (size < sizeof(tl_circle_t::TL_TAG) + sizeof(tl_circle_t))
      [[unlikely]] return false;
    memcpy(buf, &tl_circle_t::TL_TAG, sizeof(tl_circle_t::TL_TAG));
    buf += sizeof(tl_circle_t::TL_TAG);
    size -= sizeof(tl_circle_t::TL_TAG);
    return cur_circle.tl_store(buf, size);
  }

 bool operator()(const tl_rectangle_t& cur_rectanlge) noexcept {
    if (size < sizeof(tl_rectangle _t::TL_TAG) + sizeof(tl_rectangle _t))
      [[unlikely]] return false;
    memcpy(buf, &tl_rectangle _t::TL_TAG, sizeof(tl_rectangle _t::TL_TAG));
    buf += sizeof(tl_rectangle _t::TL_TAG);
    size -= sizeof(tl_rectangle _t::TL_TAG);
    return cur_rectangle.tl_store(buf, size);
  }

  uint8_t*& buf;
  size_t& size;
};

if (!std::visit(visitor_t{.buf = buf}, .size = size}, cur_figure))
  [[unlikely]] return false;
```

Алгебраические типы данных: запись

- Пишем тег сериализуемой альтернативы

```
struct visitor_t {
  bool operator()(const tl_circle_t& cur_circle) noexcept {
    if (size < sizeof(tl_circle_t::TL_TAG) + sizeof(tl_circle_t))
      [[unlikely]] return false;
    memcpy(buf, &tl_circle_t::TL_TAG, sizeof(tl_circle_t::TL_TAG));
    buf += sizeof(tl_circle_t::TL_TAG);
    size -= sizeof(tl_circle_t::TL_TAG);
    return cur_circle.tl_store(buf, size);
  }

 bool operator()(const tl_rectangle_t& cur_rectanlge) noexcept {
    if (size < sizeof(tl_rectangle _t::TL_TAG) + sizeof(tl_rectangle _t))
      [[unlikely]] return false;
    memcpy(buf, &tl_rectangle _t::TL_TAG, sizeof(tl_rectangle _t::TL_TAG));
    buf += sizeof(tl_rectangle _t::TL_TAG);
    size -= sizeof(tl_rectangle _t::TL_TAG);
    return cur_rectangle.tl_store(buf, size);
  }

  uint8_t*& buf;
  size_t& size;
};

if (!std::visit(visitor_t{.buf = buf}, .size = size}, cur_figure))
  [[unlikely]] return false;
```

Алгебраические типы данных: запись

- Сразу за тегом сериализуем тело альтернативы

```
struct visitor_t {
  bool operator()(const tl_circle_t& cur_circle) noexcept {
    if (size < sizeof(tl_circle_t::TL_TAG) + sizeof(tl_circle_t))
      [[unlikely]] return false;
    memcpy(buf, &tl_circle_t::TL_TAG, sizeof(tl_circle_t::TL_TAG));
    buf += sizeof(tl_circle_t::TL_TAG);
    size -= sizeof(tl_circle_t::TL_TAG);
    return cur_circle.tl_store(buf, size);
  }

 bool operator()(const tl_rectangle_t& cur_rectanlge) noexcept {
    if (size < sizeof(tl_rectangle _t::TL_TAG) + sizeof(tl_rectangle _t))
      [[unlikely]] return false;
    memcpy(buf, &tl_rectangle _t::TL_TAG, sizeof(tl_rectangle _t::TL_TAG));
    buf += sizeof(tl_rectangle _t::TL_TAG);
    size -= sizeof(tl_rectangle _t::TL_TAG);
    return cur_rectangle.tl_store(buf, size);
  }

  uint8_t*& buf;
  size_t& size;
};

if (!std::visit(visitor_t{.buf = buf}, .size = size}, cur_figure))
  [[unlikely]] return false;
```

Алгебраические типы данных: чтение

- Читаем тег

```
uint32_t tag;
if (size < sizeof(tag))
  [[unlikely]] return false;
memcpy(&tag, buf, sizeof(tag));
buf += sizeof(tag);
size -= sizeof(tag);
if (tag == tl_circle_t::TL_TAG) {
  tl_circle_t circle{};
  if (!circle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(circle);
} else if (tag == tl_rectangle_t::TL_TAG) {
  tl_rectangle_t rectangle{};
  if (! rectangle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(rectangle);
} else [[unlikely]] return false;
```

Алгебраические типы данных: чтение

- По тегу понимаем, какую из альтернатив будем читать дальше
- Читаем нужную альтернативу

```
uint32_t tag;
if (size < sizeof(tag))
  [[unlikely]] return false;
memcpy(&tag, buf, sizeof(tag));
buf += sizeof(tag);
size -= sizeof(tag);
if (tag == tl_circle_t::TL_TAG) {
  tl_circle_t circle{};
  if (!circle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(circle);
} else if (tag == tl_rectangle_t::TL_TAG) {
  tl_rectangle_t rectangle{};
  if (! rectangle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(rectangle);
} else [[unlikely]] return false;
```

Алгебраические типы данных: чтение

- Читаем тег

По тегу понимаем, какую из альтернатив будем читать дальше

- Читаем нужную альтернативу

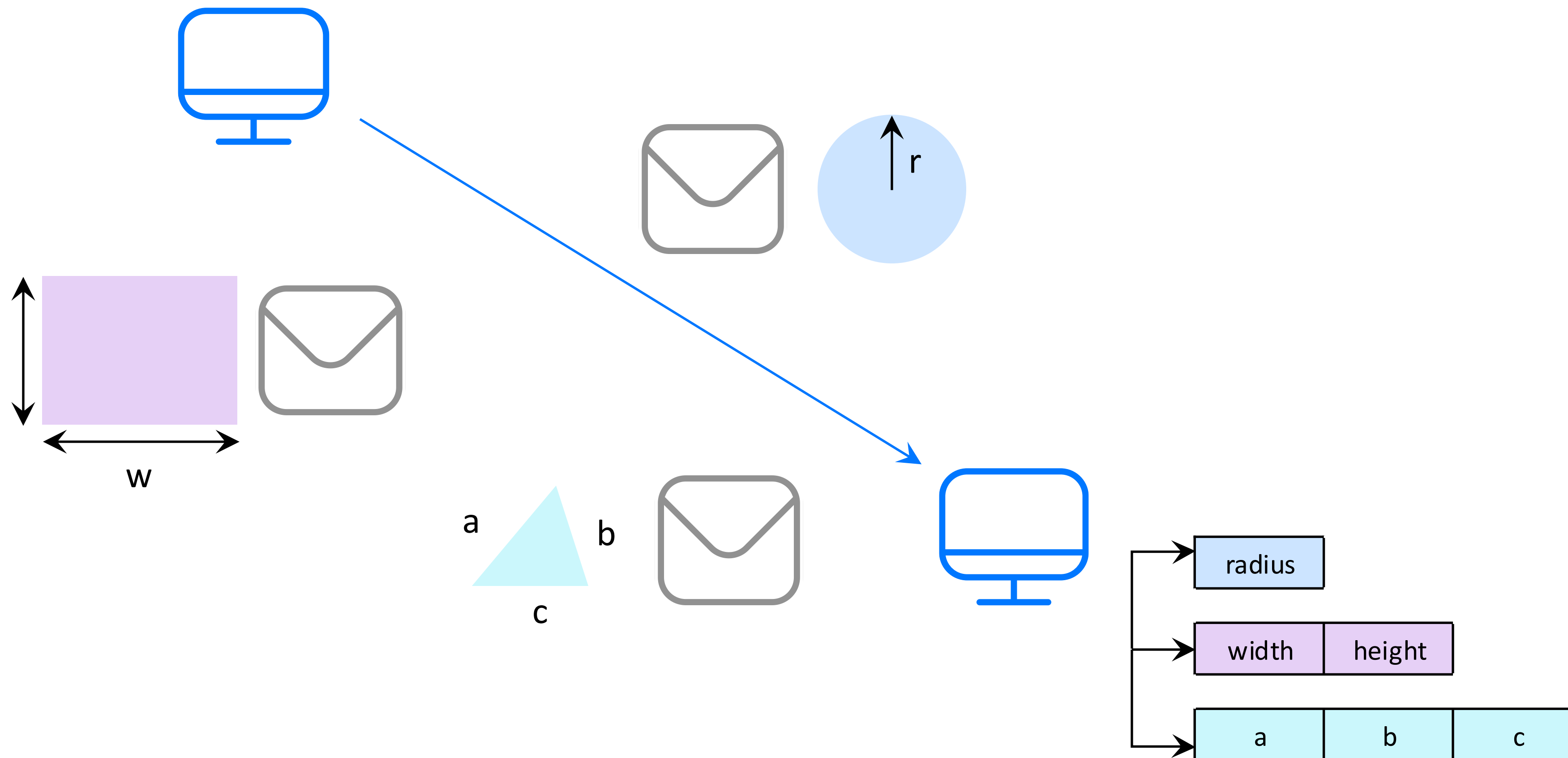
```
uint32_t tag;
if (size < sizeof(tag))
  [[unlikely]] return false;
memcpy(&tag, buf, sizeof(tag));
buf += sizeof(tag);
size -= sizeof(tag);
if (tag == tl_circle_t::TL_TAG) {
  tl_circle_t circle{};
  if (!circle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(circle);
} else if (tag == tl_rectangle_t::TL_TAG) {
  tl_rectangle_t rectangle{};
  if (! rectangle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(rectangle);
} else [[unlikely]] return false;
```

Алгебраические типы данных: чтение

- Отвечаем ошибкой если тег не совпал ни с одним из ожидаемых

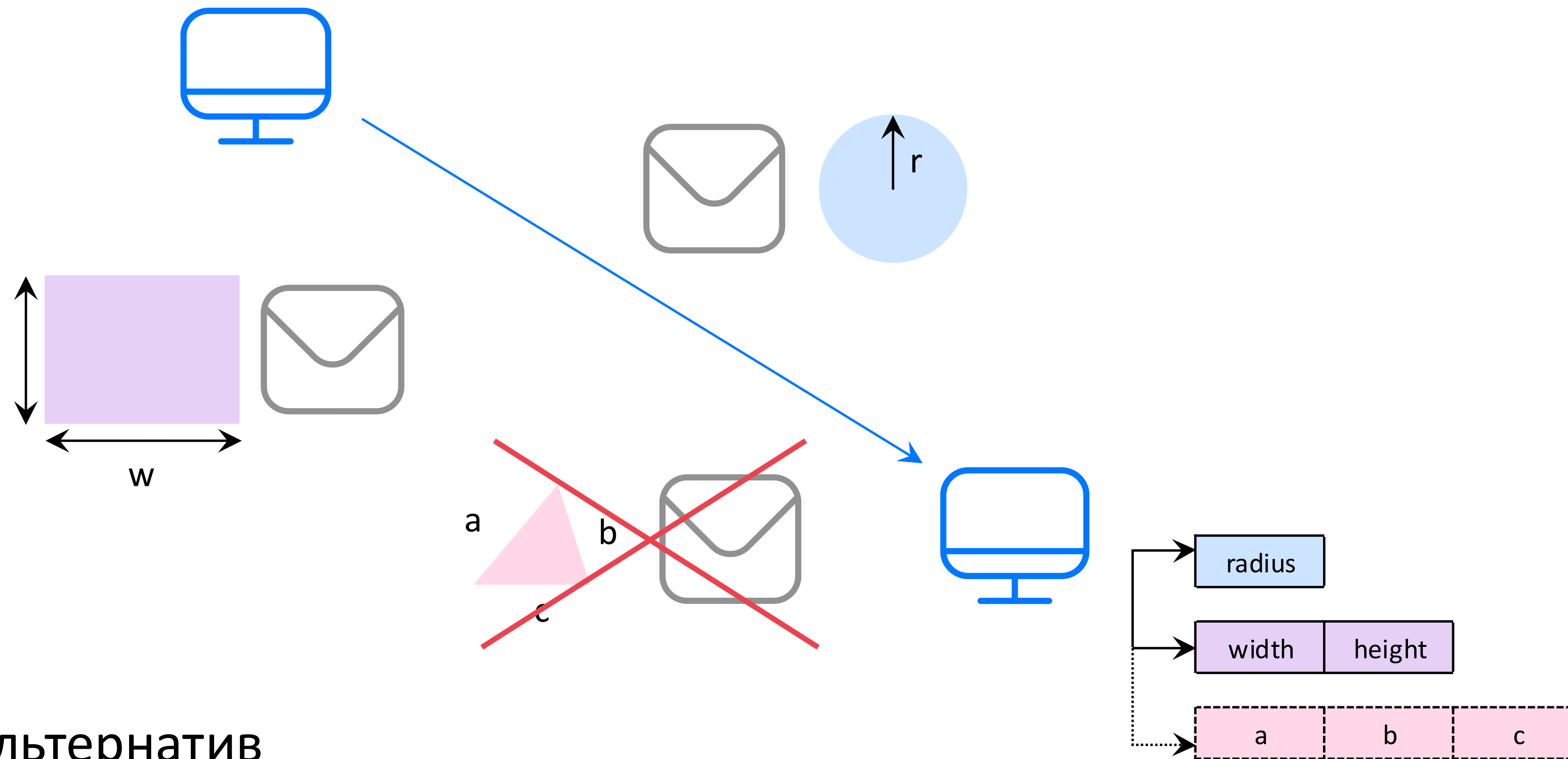
```
uint32_t tag;
if (size < sizeof(tag))
  [[unlikely]] return false;
memcpy(&tag, buf, sizeof(tag));
buf += sizeof(tag);
size -= sizeof(tag);
if (tag == tl_circle_t::TL_TAG) {
  tl_circle_t circle{};
  if (!circle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(circle);
} else if (tag == tl_rectangle_t::TL_TAG) {
  tl_rectangle_t rectangle{};
  if (! rectangle.tl_fetch(buf, size))
    [[unlikely]] return false;
  figures.emplace_back(rectangle);
} else [[unlikely]] return false;
```

Обновление схемы с АТД



Есть протокол для добавления новых альтернатив…

Обновление схемы с АТД



… и для удаления
неиспользуемых альтернатив

Бенчмарки работы с ADT

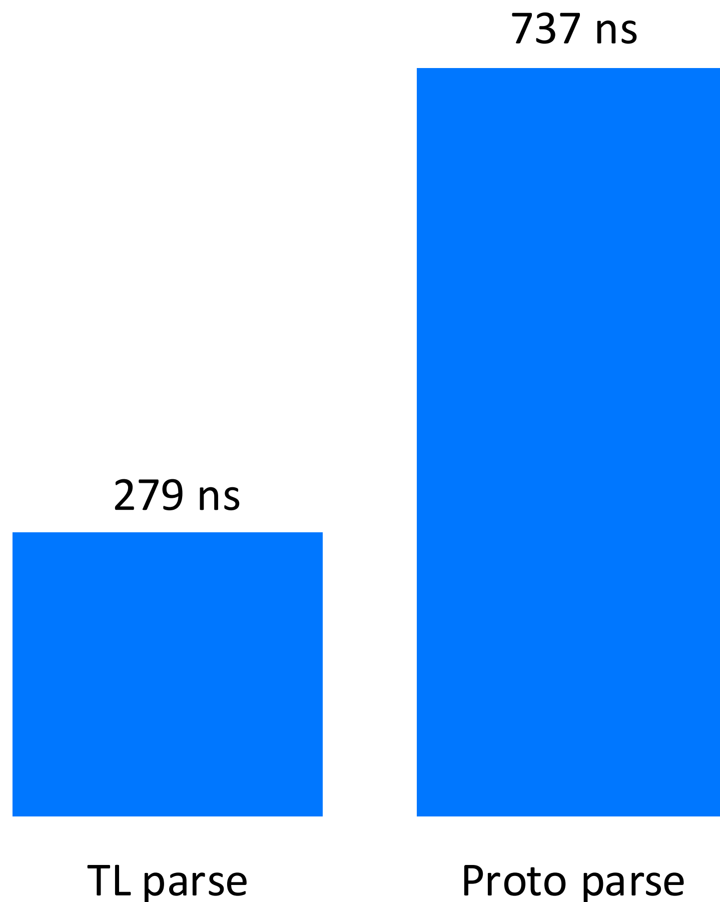
```
circle#00123456
  radius: int
= Figure;

rectangle#00abcdef
  width: int
  height: int
= Figure;

figures
  figures: vector Figure
= Figures;
```

```cpp
static void BM_TL_adt_store(benchmark::State& state) {
    // 50 circles, 50 rectangles
    tl_figures_t from{prepare_tl()};
    std::array<uint8_t, 1024 * 1024> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = from.tl_store(buf.data(),
                                    buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_adt_store(benchmark::State& state) {
    // 50 circles, 50 rectangles
    protobench::Figures from{prepare_proto()};
    std::array<uint8_t, 1024 * 1024> buf{};
    for ([[maybe_unused]] const auto _ : state) {
        from.SerializeToArray(buf.data(),
                              buf.size());
        benchmark::DoNotOptimize(buf);
        benchmark::ClobberMemory();
    }
}
```

```protobuf
message Circle {
  int32 radius = 1;
}

message Rectangle {
  int32 width = 1;
  int32 height = 2;
}

message Figure {
  oneof figure {
    Circle circle = 1;
    Rectangle rectangle = 2;
  }
}

message Figures {
  repeated Figure figures = 1;
}
```

Бенчмарки работы с ADT

```
circle#00123456
  radius: int
= Figure;

rectangle#00abcdef
  width: int
  height: int
= Figure;

figures
  figures: vector Figure
= Figures;
```




```
message Circle {
  int32 radius = 1;
}

message Rectangle {
  int32 width = 1;
  int32 height = 2;
}

message Figure {
  oneof figure {
    Circle circle = 1;
    Rectangle rectangle = 2;
  }
}

message Figures {
  repeated Figure figures = 1;
}
```

Бенчмарки работы с ADT

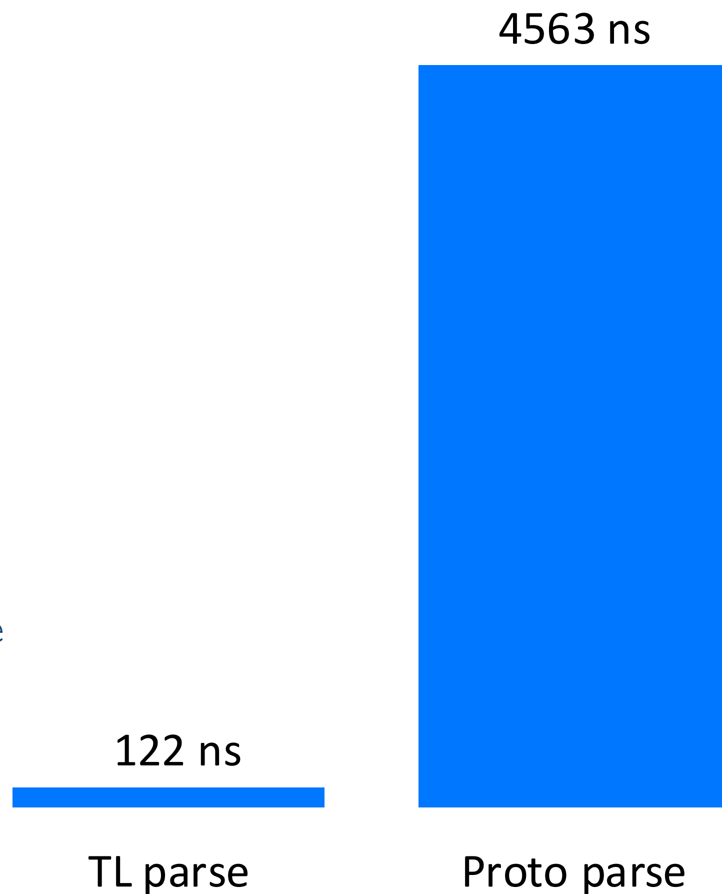
```
circle#00123456
  radius: int
= Figure;

rectangle#00abcdef
  width: int
  height: int
= Figure;

figures
  figures: vector Figure
= Figures;
```

```
static void BM_TL_adt_parse(benchmark::State& state) {
    // 50 circles, 50 rectangles
    tl_figures_t from{prepare_tl()};
    std::array<uint8_t, 1024 * 1024> buf{};
    std::ignore = from.tl_store(buf.data(), buf.size());
    tl_figures_t to{};
    to.figures.reserve(128);
    for ([[maybe_unused]] const auto _ : state) {
        std::ignore = to.tl_fetch(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}

static void BM_proto_adt_parse(benchmark::State& state) {
    // 50 circles, 50 rectangles
    protobench::Figures from{prepare_proto()};
    std::array<uint8_t, 1024 * 1024> buf{};
    from.SerializeToArray(buf.data(), buf.size());
    protobench::Figures to{};
    to.mutable_figures()->Reserve(128);
    for ([[maybe_unused]] const auto _ : state) {
        to.ParseFromArray(buf.data(), buf.size());
        benchmark::DoNotOptimize(to);
        benchmark::ClobberMemory();
    }
}
```

```
message Circle {
  int32 radius = 1;
}

message Rectangle {
  int32 width = 1;
  int32 height = 2;
}

message Figure {
  oneof figure {
    Circle circle = 1;
    Rectangle rectangle = 2;
  }
}

message Figures {
  repeated Figure figures = 1;
}
```

Бенчмарки работы с ADT

```
circle#00123456
  radius: int
= Figure;

rectangle#00abcdef
  width: int
  height: int
= Figure;

figures
  figures: vector Figure
= Figures;
```




```
message Circle {
  int32 radius = 1;
}

message Rectangle {
  int32 width = 1;
  int32 height = 2;
}

message Figure {
  oneof figure {
    Circle circle = 1;
    Rectangle rectangle = 2;
  }
}

message Figures {
  repeated Figure figures = 1;
}
```

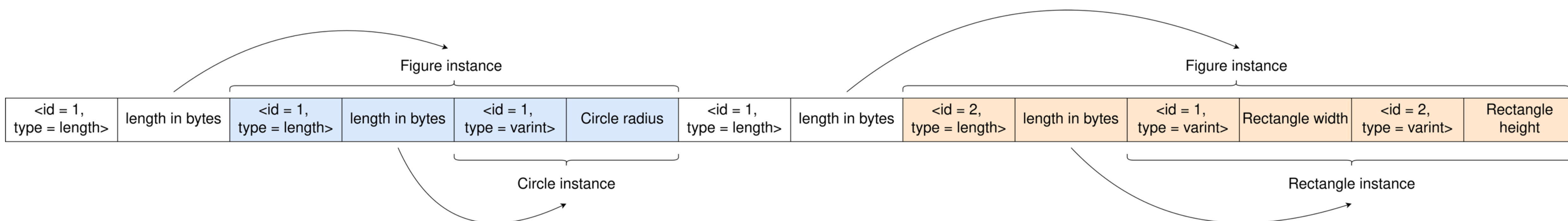
ADT в Protobuf

```
message Circle {
  int32 radius = 1;
}
```

```
message Rectangle {
  int32 width = 1;
  int32 height = 2;
}
```

- Для каждого круга/прямоугольника два набора служебной информации
  - Для обозначения repeated-поля
  - Для обозначения oneof-поля
- Не считая служебной информации для полей круга и квадрата

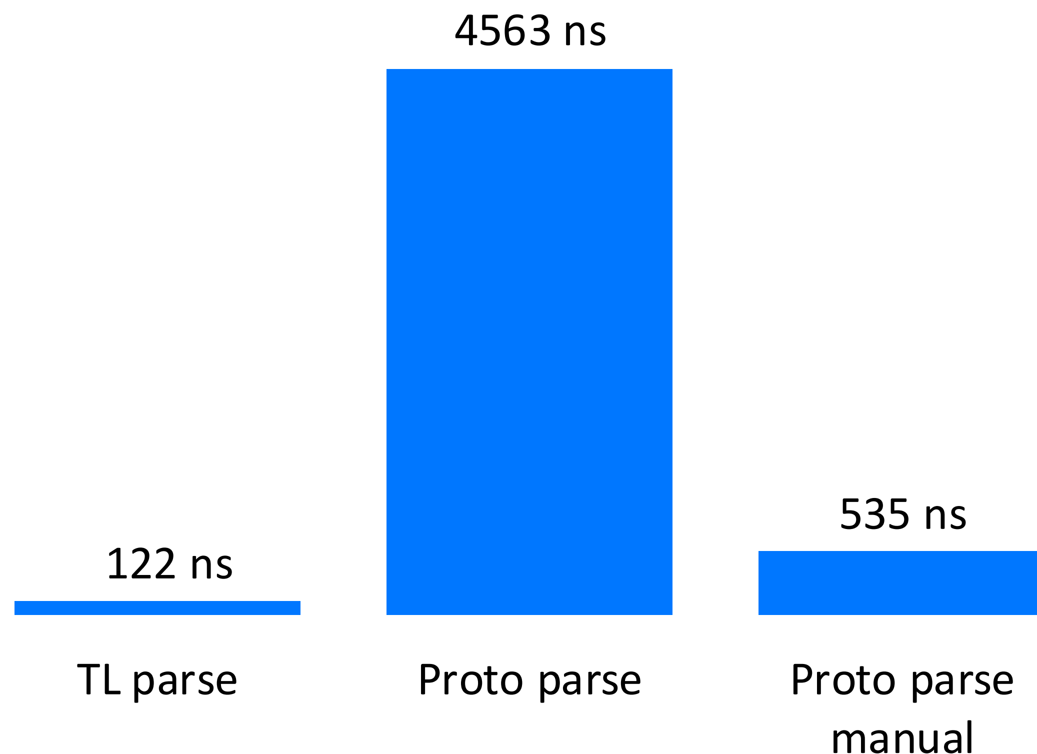
```
message Figure {
  oneof figure {
    Circle circle = 1;
    Rectangle rectangle = 2;
  }
}

message Figures {
  repeated Figure figures = 1;
}
```



ADT в Protobuf

```
while (true) {
  const auto tag = input.ReadTagNoLastTag();
  if (tag == 0) [[unlikely]] break;
  if (tag != 10u) [[unlikely]] std::abort();
  int32_t length_unused;
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  const auto figure_type = input.ReadTagNoLastTag();
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  if (figure_type == 10u) { // circle
    proto_circle_t circle{};
    const auto circle_tag = input.ReadTagNoLastTag();
    if (circle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &circle.radius)))
       [[unlikely]] std::abort();
    figures.emplace_back(circle);
  } else if (figure_type == 18u) { // rectangle
    proto_rectangle_t rectangle{};
    const auto rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.width)))
       [[unlikely]] std::abort();
    rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 16u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.height)))
       [[unlikely]] std::abort();
    figures.emplace_back(rectangle);
  } else [[unlikely]] std::abort();
}
```

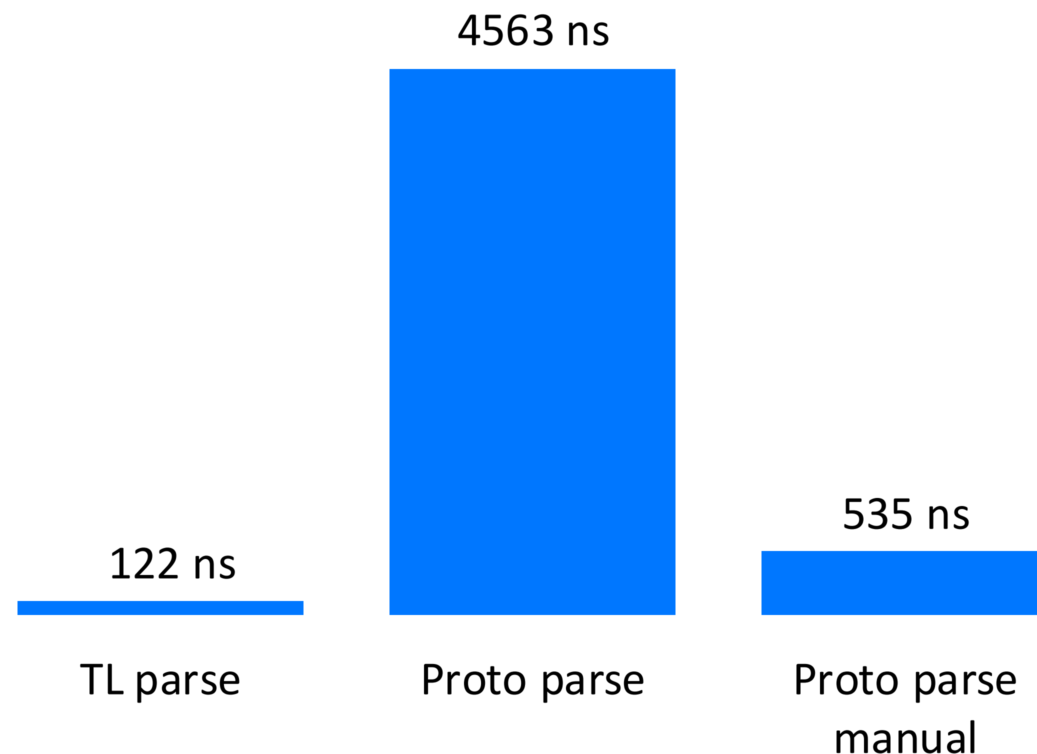
Разобрав структуру Protobuf руками, получаем выигрыш почти на порядок



ADT в Protobuf

```
while (true) {
  const auto tag = input.ReadTagNoLastTag();
  if (tag == 0) [[unlikely]] break;
  if (tag != 10u) [[unlikely]] std::abort();
  int32_t length_unused;
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  const auto figure_type = input.ReadTagNoLastTag();
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  if (figure_type == 10u) { // circle
    proto_circle_t circle{};
    const auto circle_tag = input.ReadTagNoLastTag();
    if (circle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &circle.radius)))
       [[unlikely]] std::abort();
    figures.emplace_back(circle);
  } else if (figure_type == 18u) { // rectangle
    proto_rectangle_t rectangle{};
    const auto rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.width)))
       [[unlikely]] std::abort();
    rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 16u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.height)))
       [[unlikely]] std::abort();
    figures.emplace_back(rectangle);
  } else [[unlikely]] std::abort();
}
```

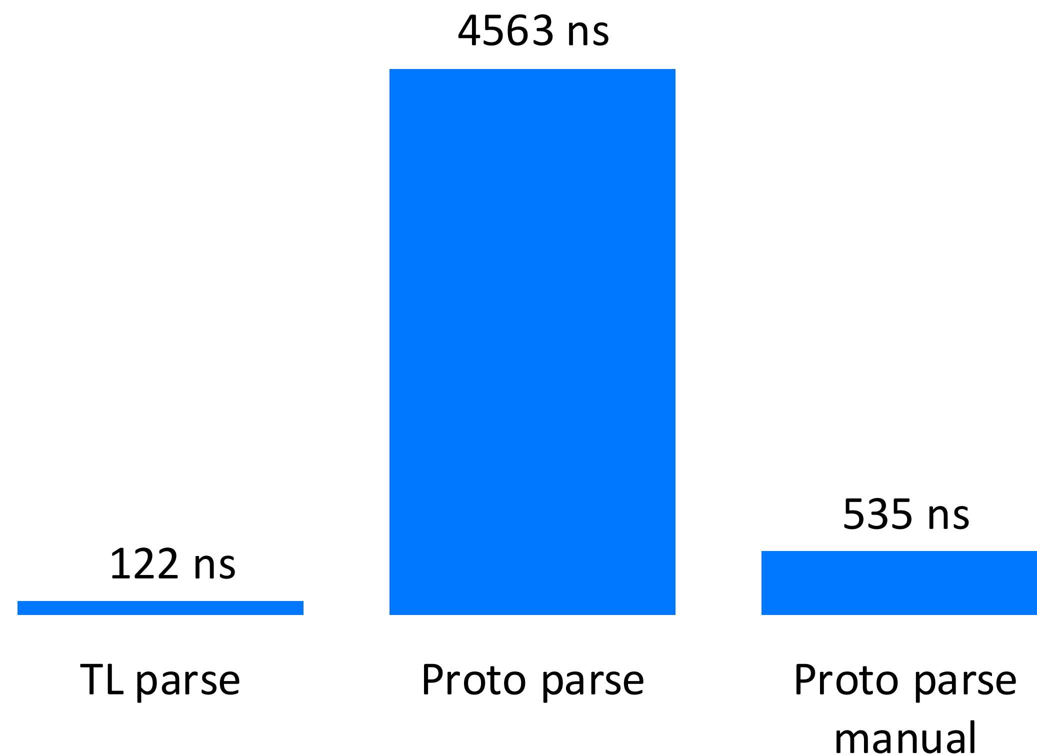
Разобрав структуру Protobuf руками, получаем выигрыш почти на порядок



ADT в Protobuf

```
while (true) {
  const auto tag = input.ReadTagNoLastTag();
  if (tag == 0) [[unlikely]] break;
  if (tag != 10u) [[unlikely]] std::abort();
  int32_t length_unused;
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  const auto figure_type = input.ReadTagNoLastTag();
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  if (figure_type == 10u) { // circle
    proto_circle_t circle{};
    const auto circle_tag = input.ReadTagNoLastTag();
    if (circle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &circle.radius)))
       [[unlikely]] std::abort();
    figures.emplace_back(circle);
  } else if (figure_type == 18u) { // rectangle
    proto_rectangle_t rectangle{};
    const auto rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.width)))
       [[unlikely]] std::abort();
    rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 16u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.height)))
       [[unlikely]] std::abort();
    figures.emplace_back(rectangle);
  } else [[unlikely]] std::abort();
}
```

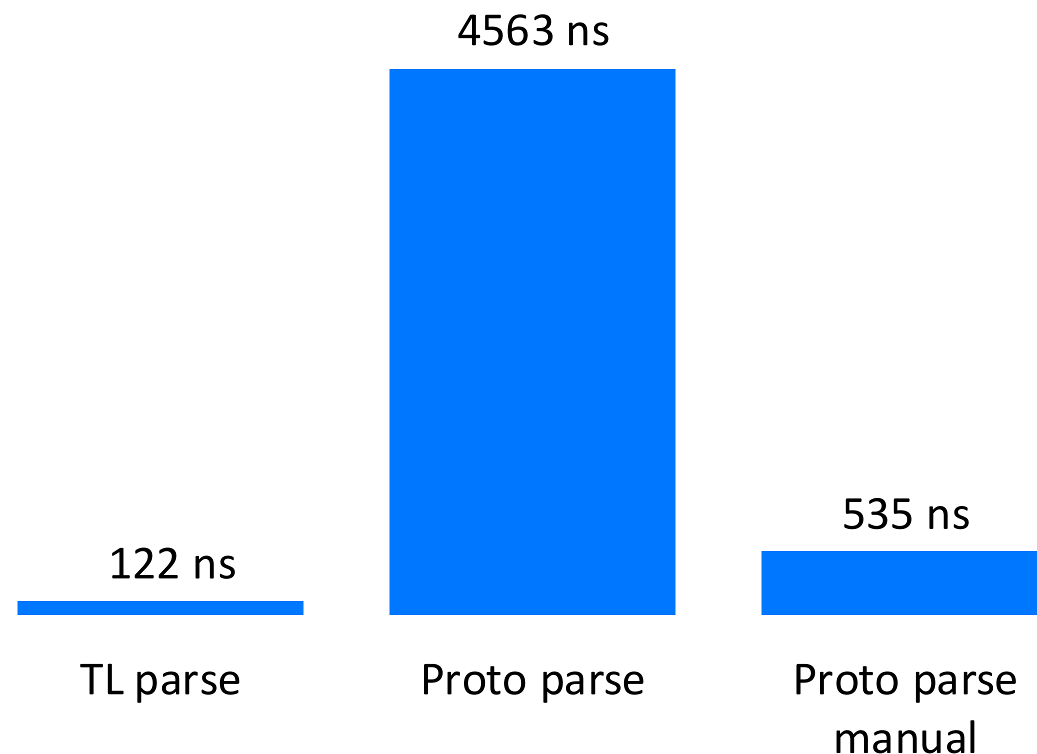
Разобрав структуру Protobuf руками, получаем выигрыш почти на порядок



ADT в Protobuf

```
while (true) {
  const auto tag = input.ReadTagNoLastTag();
  if (tag == 0) [[unlikely]] break;
  if (tag != 10u) [[unlikely]] std::abort();
  int32_t length_unused;
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  const auto figure_type = input.ReadTagNoLastTag();
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  if (figure_type == 10u) { // circle
    proto_circle_t circle{};
    const auto circle_tag = input.ReadTagNoLastTag();
    if (circle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &circle.radius))
       [[unlikely]] std::abort();
    figures.emplace_back(circle);
  } else if (figure_type == 18u) { // rectangle
    proto_rectangle_t rectangle{};
    const auto rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.width))
       [[unlikely]] std::abort();
    rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 16u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.height))
       [[unlikely]] std::abort();
    figures.emplace_back(rectangle);
  } else [[unlikely]] std::abort();
}
```

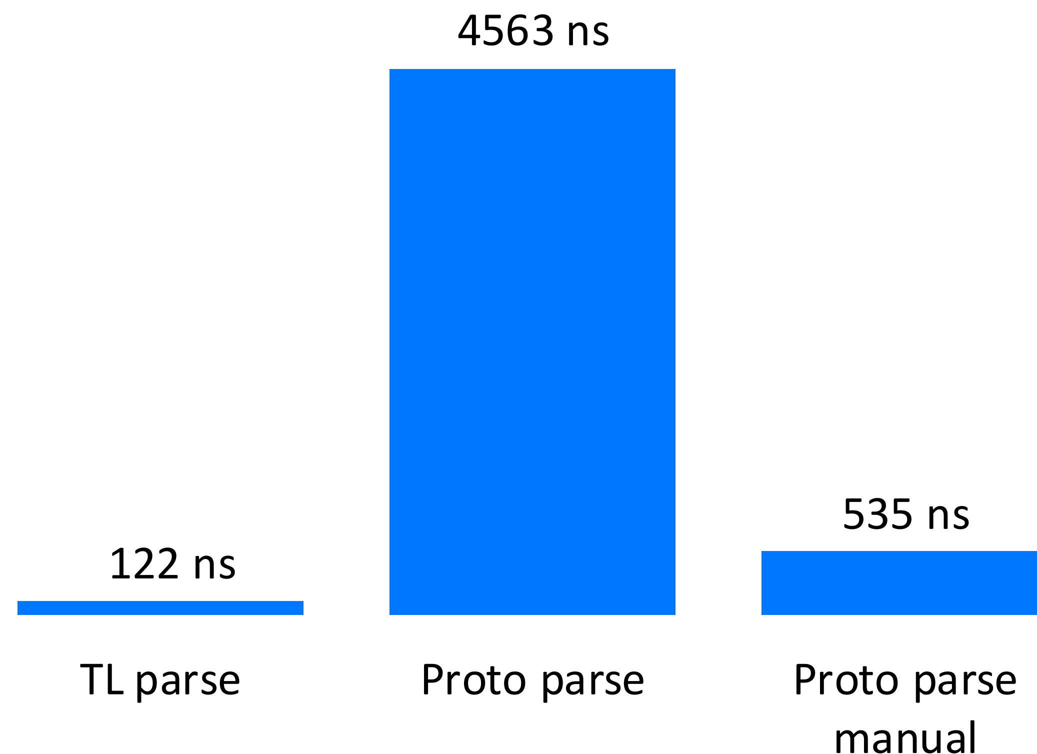
Разобрав структуру Protobuf руками, получаем выигрыш почти на порядок



ADT в Protobuf

```
while (true) {
  const auto tag = input.ReadTagNoLastTag();
  if (tag == 0) [[unlikely]] break;
  if (tag != 10u) [[unlikely]] std::abort();
  int32_t length_unused;
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  const auto figure_type = input.ReadTagNoLastTag();
  if (!input.ReadVarintSizeAsInt(&length_unused))
    [[unlikely]] std::abort();
  if (figure_type == 10u) { // circle
    proto_circle_t circle{};
    const auto circle_tag = input.ReadTagNoLastTag();
    if (circle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &circle.radius)))
      [[unlikely]] std::abort();
    figures.emplace_back(circle);
  } else if (figure_type == 18u) { // rectangle
    proto_rectangle_t rectangle{};
    const auto rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 8u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.width)))
      [[unlikely]] std::abort();
    rectangle_tag = input.ReadTagNoLastTag();
    if (rectangle_tag != 16u) [[unlikely]] std::abort();
    if (!ReadPrimitive<int32_t, TYPE_INT32>(input, &rectangle.height)))
      [[unlikely]] std::abort();
    figures.emplace_back(rectangle);
  } else [[unlikely]] std::abort();
}
```

Разобрав структуру Protobuf руками, получаем выигрыш почти на порядок



Оптимизация использования памяти

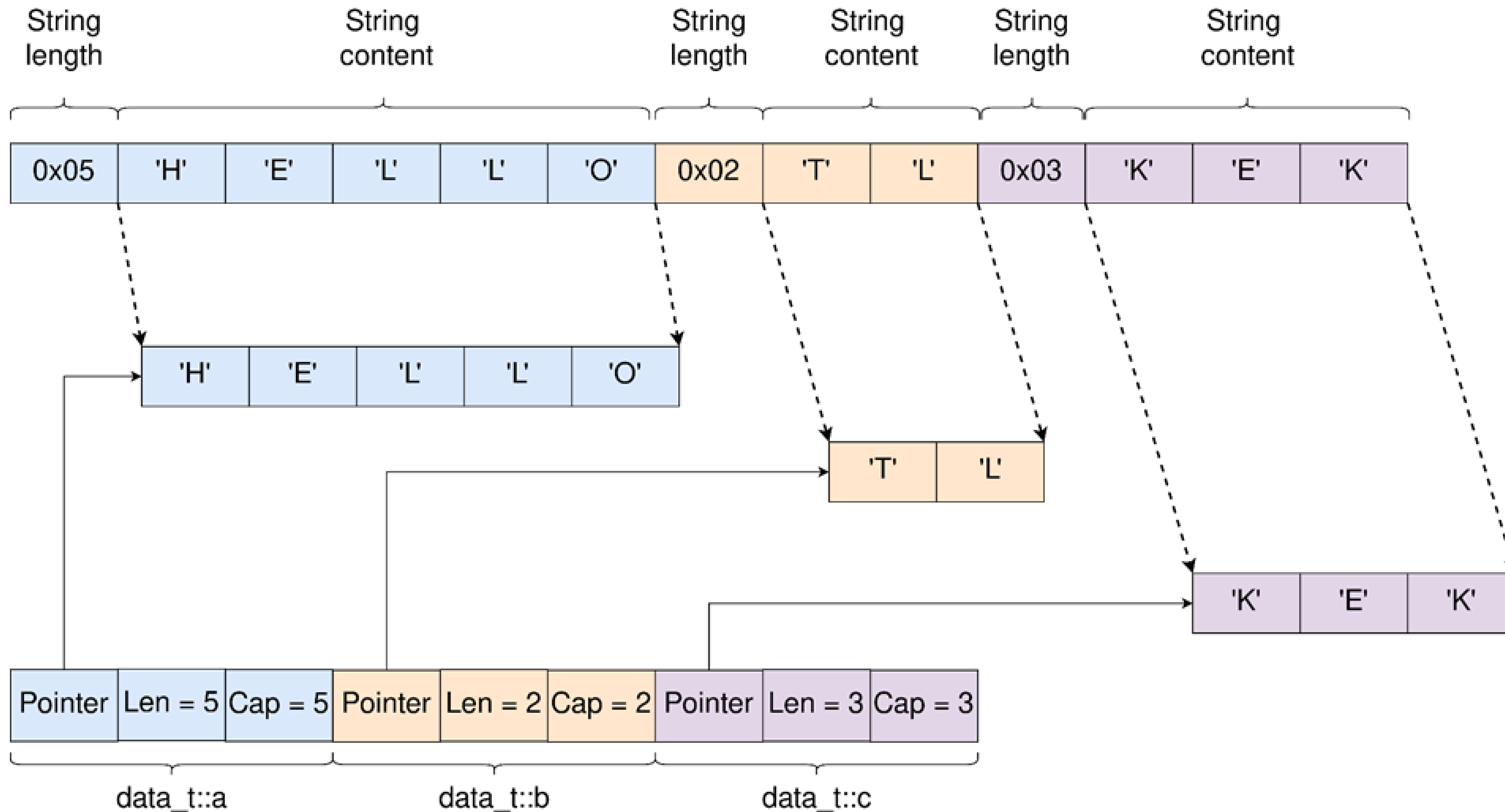
```
data a: string b: string c: string = Data;
```

- Компилируется в структуру с тремя полями типа `std::string`

```
struct data_t {
  std::string a;
  std::string b;
  std::string c;
};
```

Оптимизация использования памяти

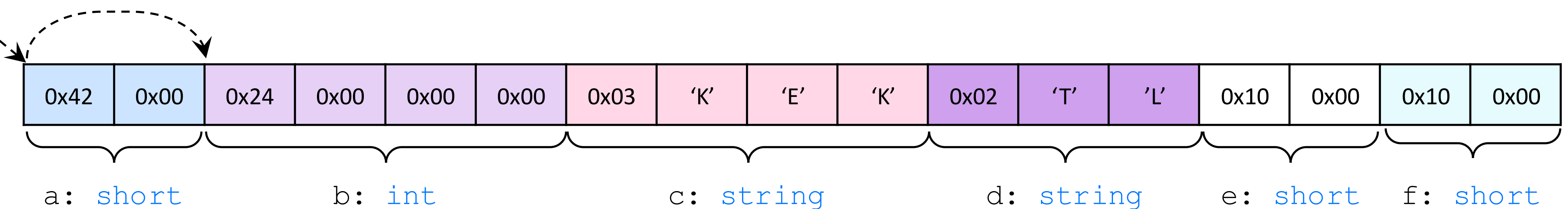
```
data a: string b: string c: string = Data;
```



Оптимизация использования памяти

Поле `a:` `short` начинается с байта#0 и занимает два байта

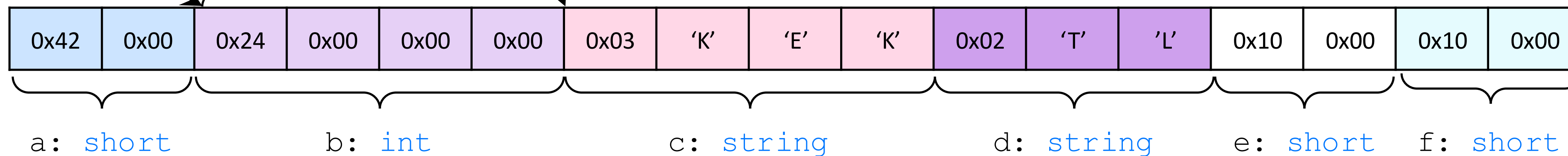
```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```



Оптимизация использования памяти

Поле `b: int` начинается с байта#(0+2) и занимает четыре байта

```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```

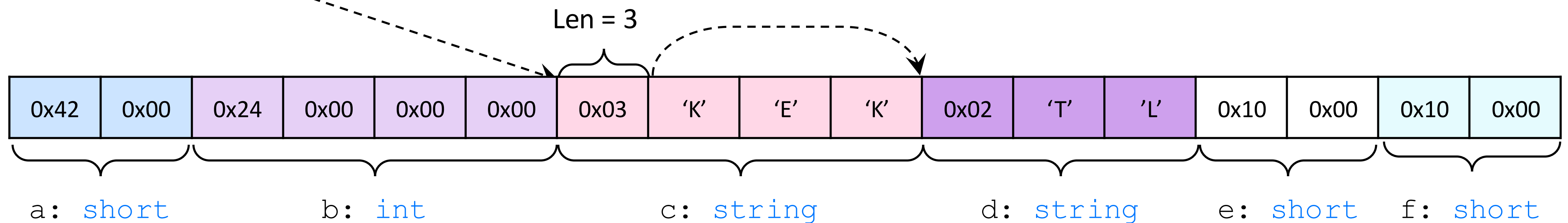


Оптимизация использования памяти

Для обращения к полю `c: string`

- Прочитать длину с байта#(0+2+4)
- Длина равна трём, занимает **один** байт
- Строка начинается с байта#(0+2+4+**1**) и занимает три байта

```
data
  a: short
  b: int
  c: string
  d: string
  e: short
  f: short
= Data;
```

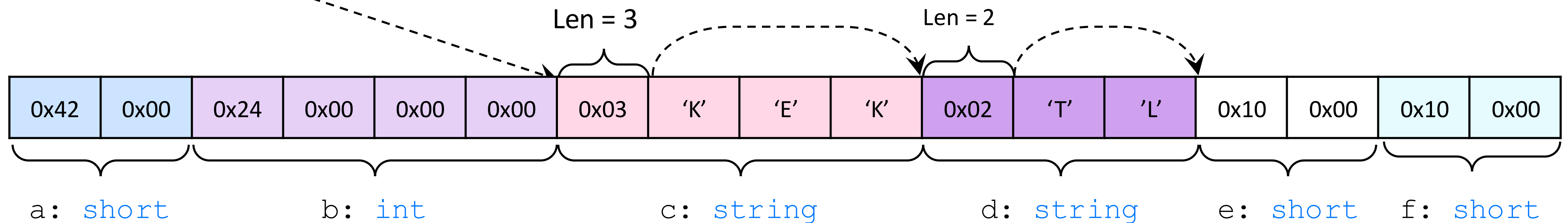


Оптимизация использования памяти

Для обращения к полю `d: string`

- Ищем адрес начала, пропуская `c: string`
- Длина равна двум и занимает один байт
- Следующие два байта — контент строки

```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```

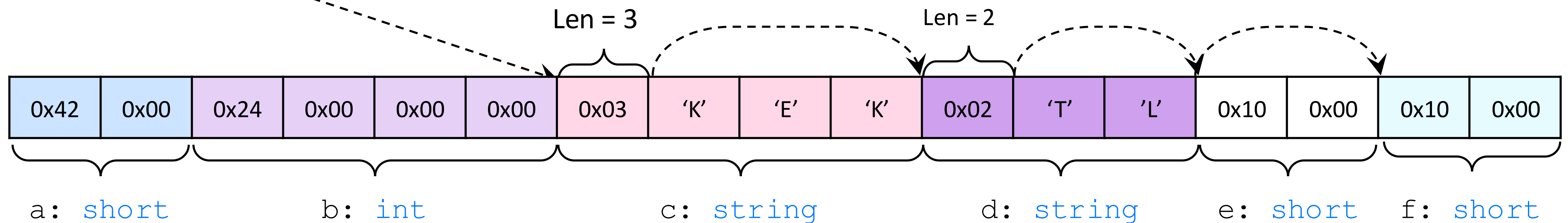


Оптимизация использования памяти

Для обращения к полю `e: short`

- Ищем адрес начала, пропуская `c: string` и `d: string`
- Читаем следующие два байта
- А если строк сто?

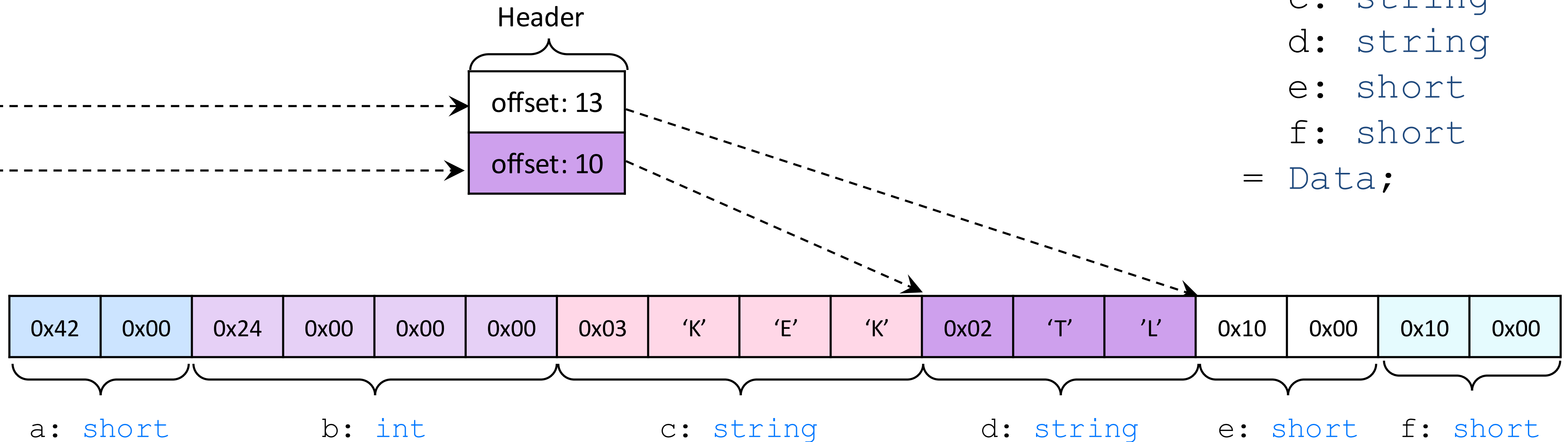
```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```



Оптимизация использования памяти

- Для каждого поля, следующего за полем не константного размера, храним адрес его начала в заголовке
- Не нужно аллоцировать и копировать

```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```

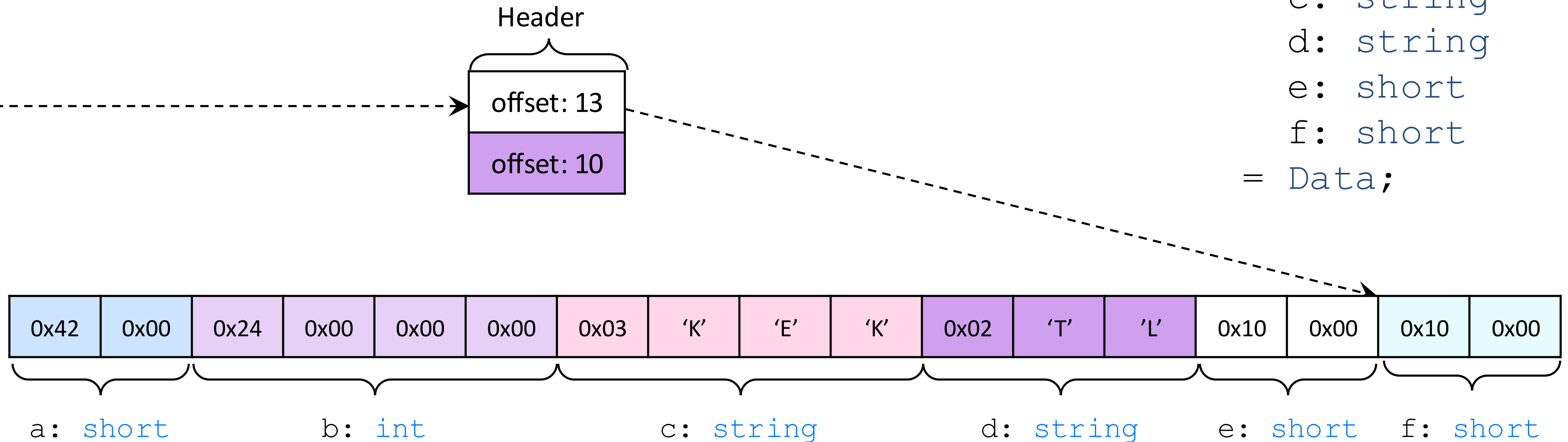


Оптимизация использования памяти

Для обращения к полю `f: short`

- Прочитать адрес начала поля `e: short`
- Прибавить к нему известный в compile-time размер поля `e: short`

```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```

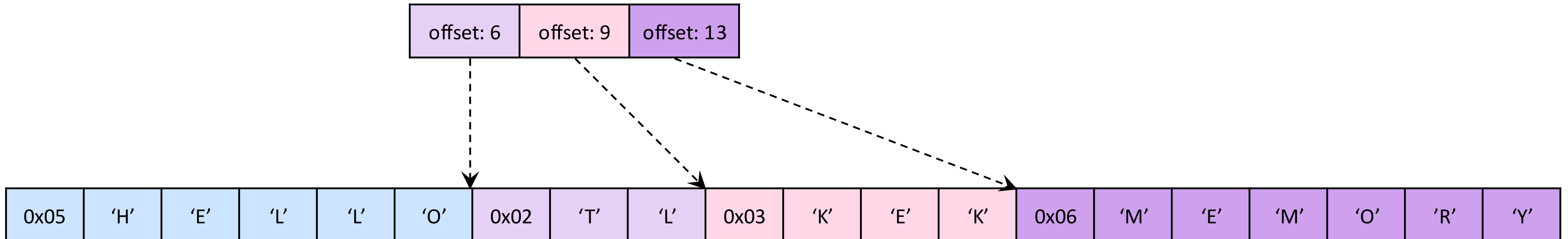


Оптимизация использования памяти

- Совсем без аллокаций не обойтись

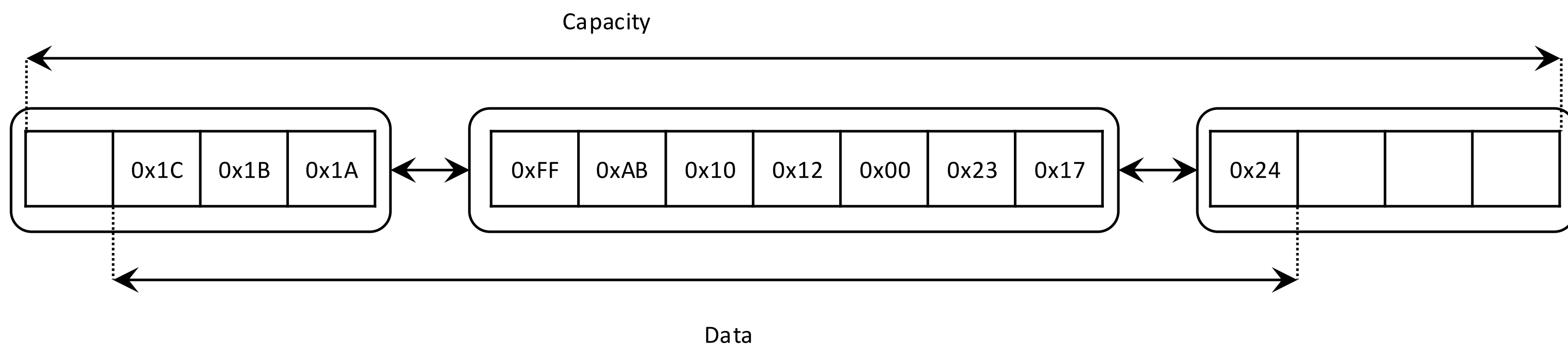
```
data lines: vector string = Data;
```

- Неизвестное заранее количество строк
- Нужно запомнить неизвестное заранее количество адресов начала
- Придётся аллоцировать массив с оффсетами



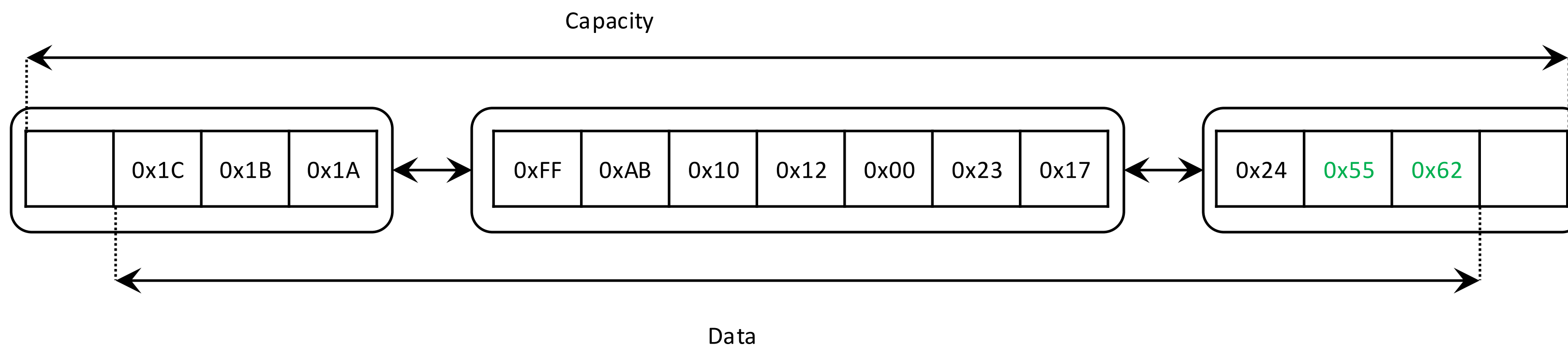
Сетевая буферизация

- Храним сетевые данные не в непрерывном куске памяти, а в цепочках блоков
- Похоже на `sk_buff`
- Ну или на внутреннее устройство `std::deque`
- Блоки могут быть разного размера



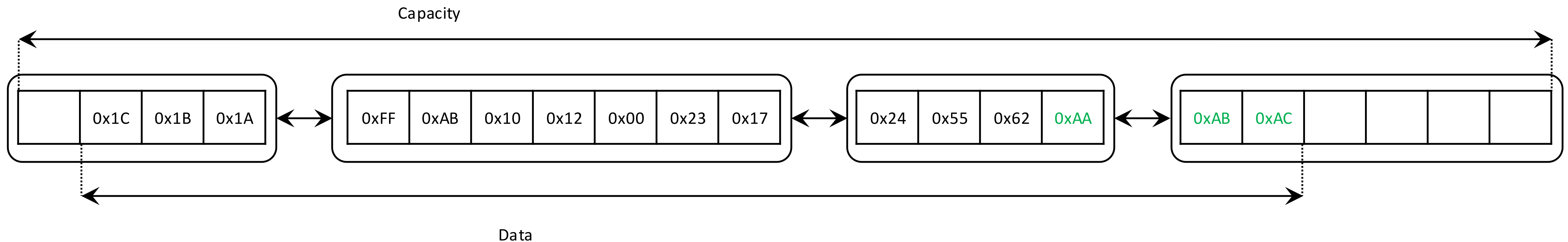
Сетевая буферизация

- Приходящие из сокета байты читаются в конец головного блока…



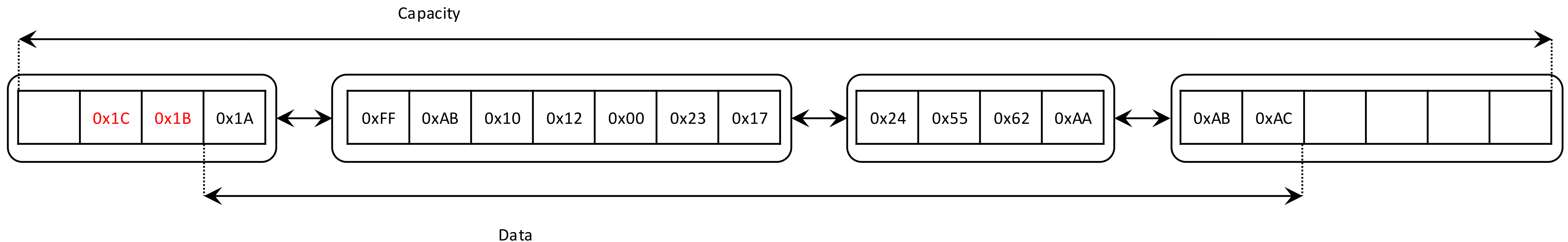
Сетевая буферизация

- … или в новый блок, если в старом не хватает места



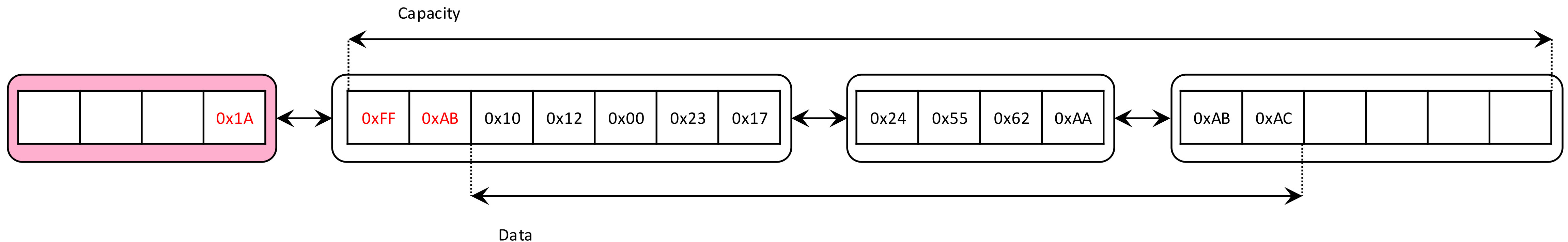
Сетевая буферизация

- Обработанные данные удаляются из конца хвостового блока



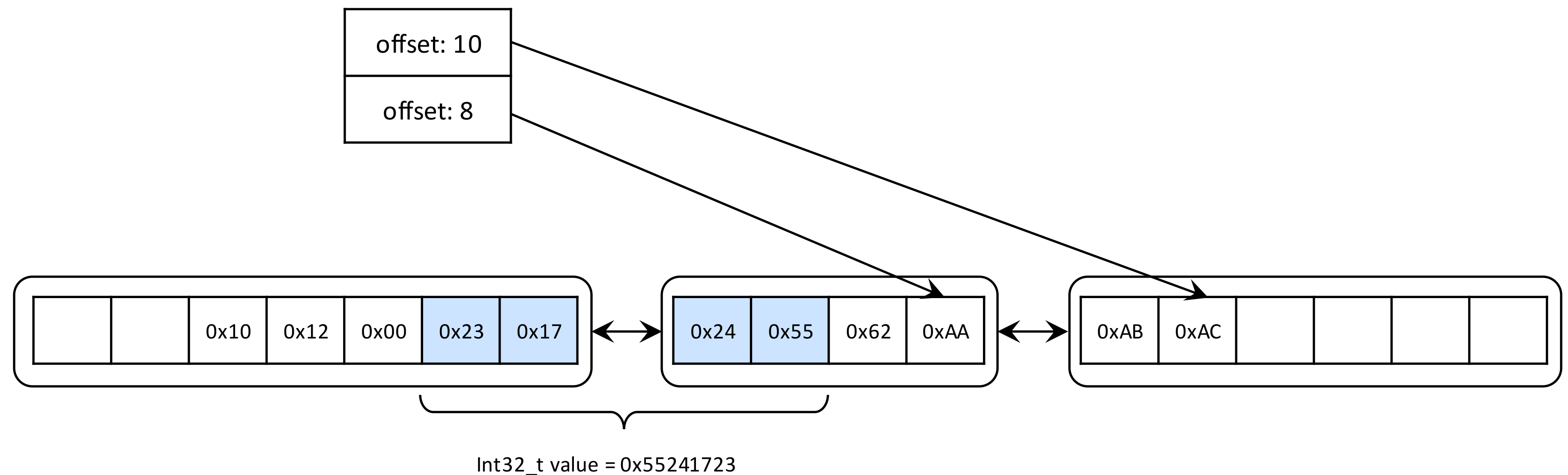
Сетевая буферизация

- Когда в хвостовом блоке не осталось необработанных данных, он может быть освобождён
- Multiple readers, concurrent memory reclamation, refcounting…



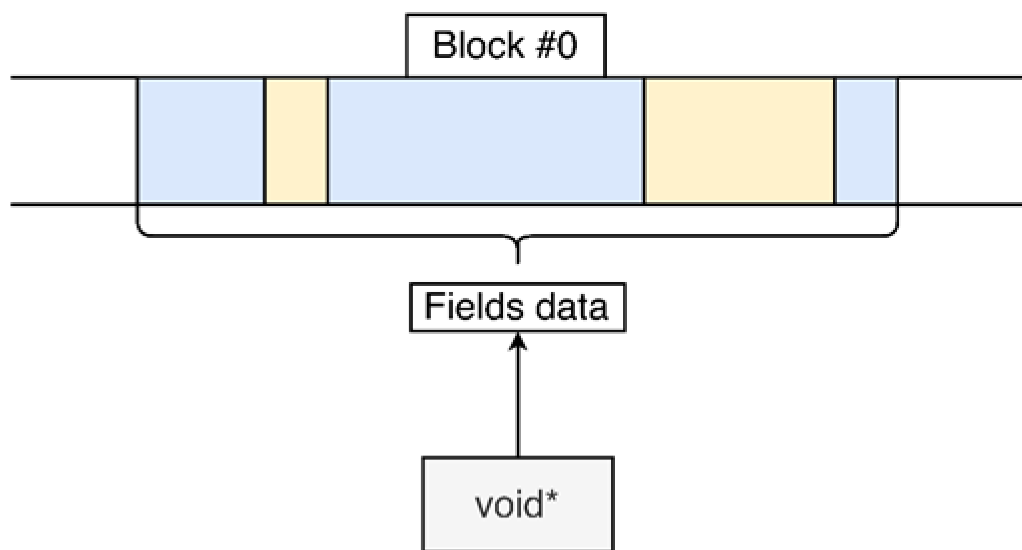
Сетевая буферизация

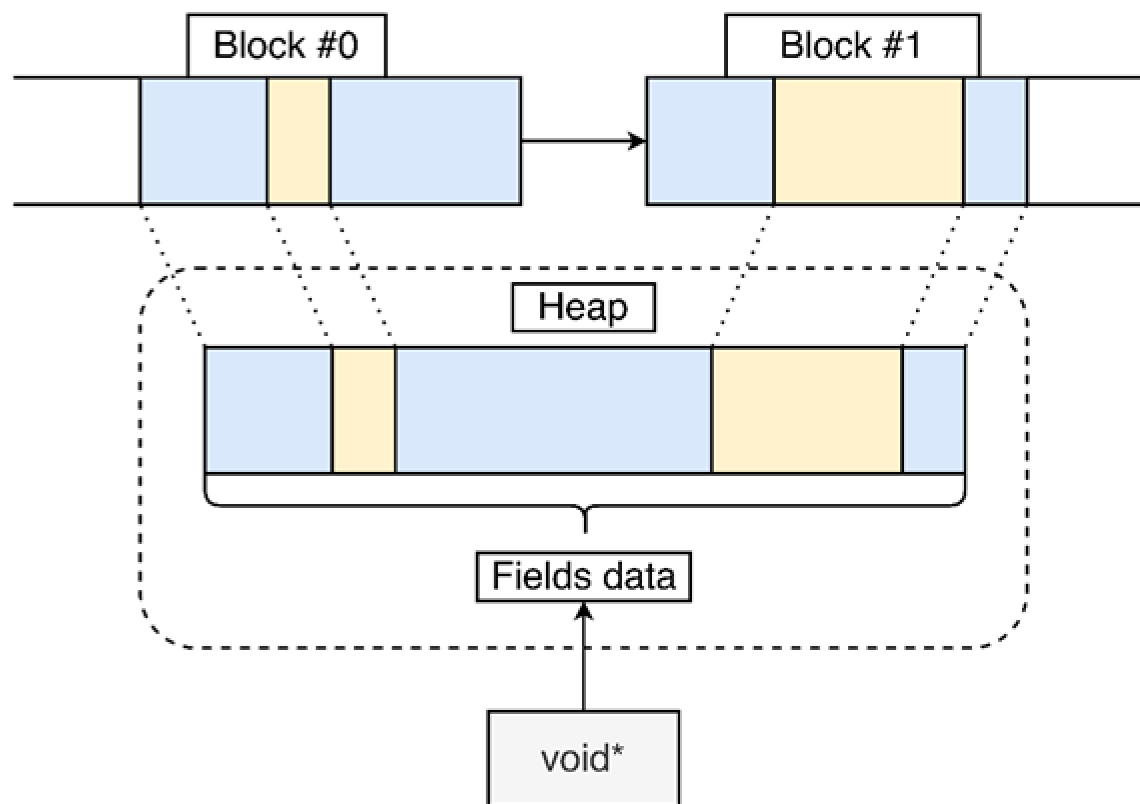
- Объект, даже примитивный, может пересекать границы буферов
- По адресу начала объекта нужно ещё понять, в каком из блоков находится объект



Реализация оптимизации

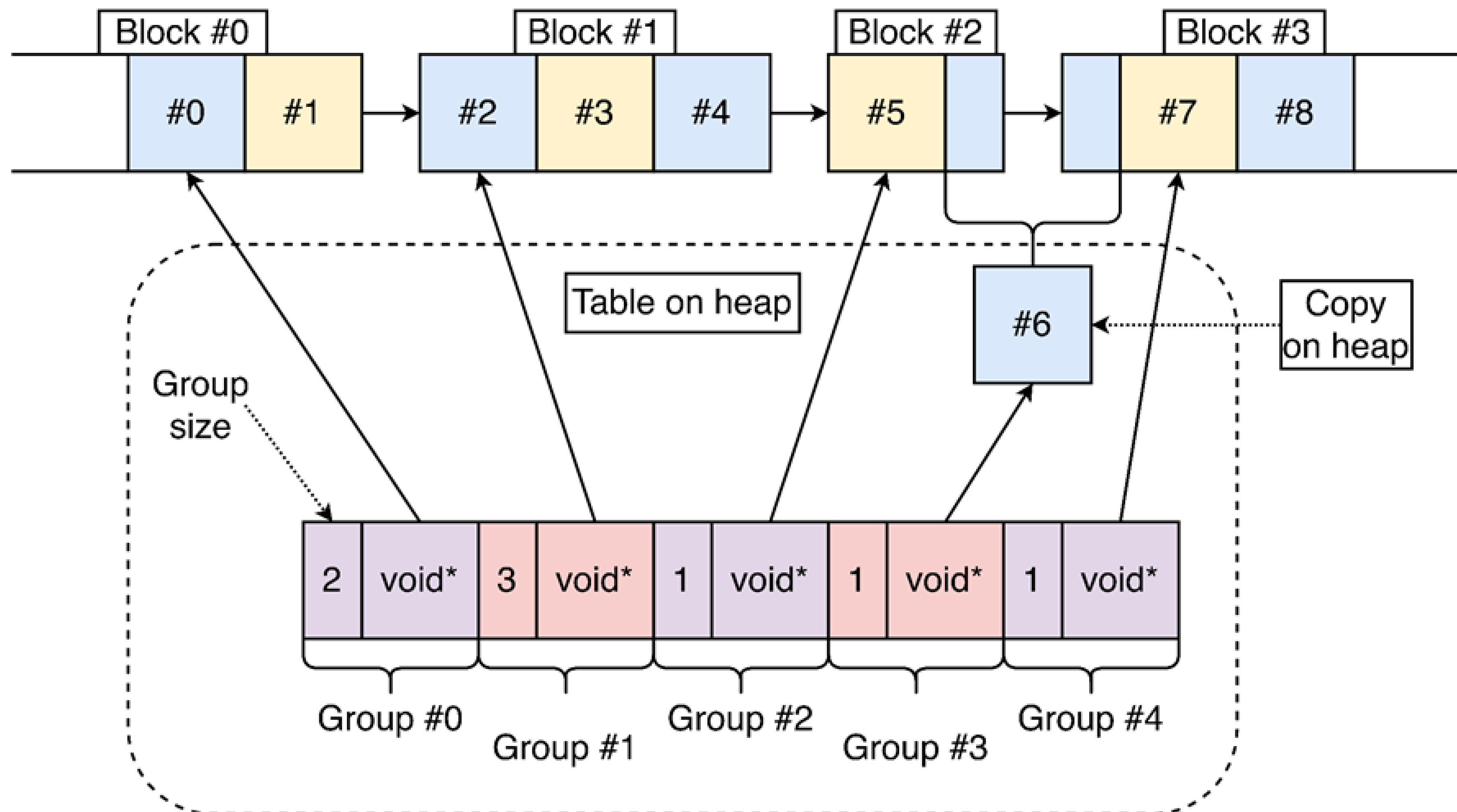
Чтение POD'ов





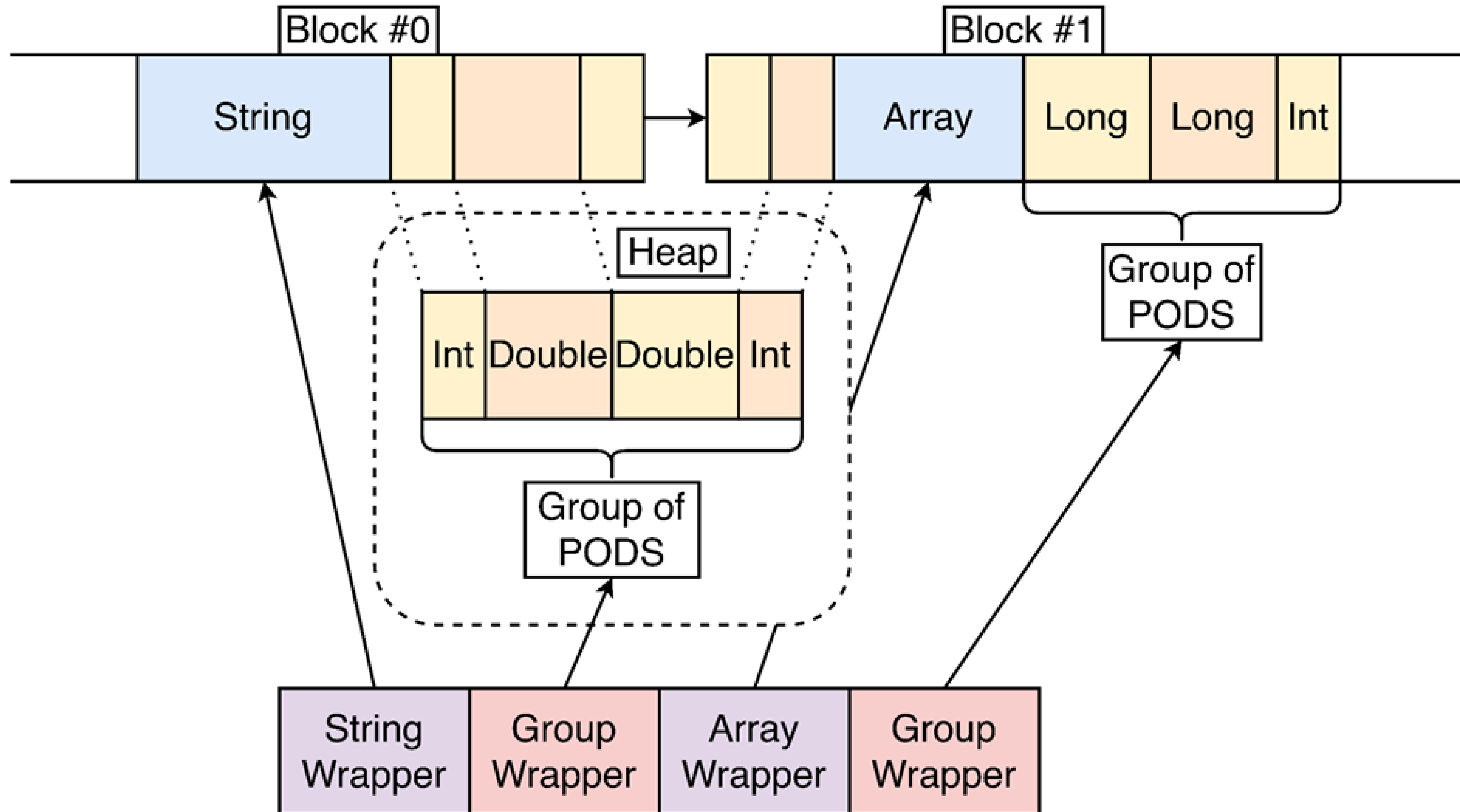
Реализация оптимизации

Чтение массивов POD’ов



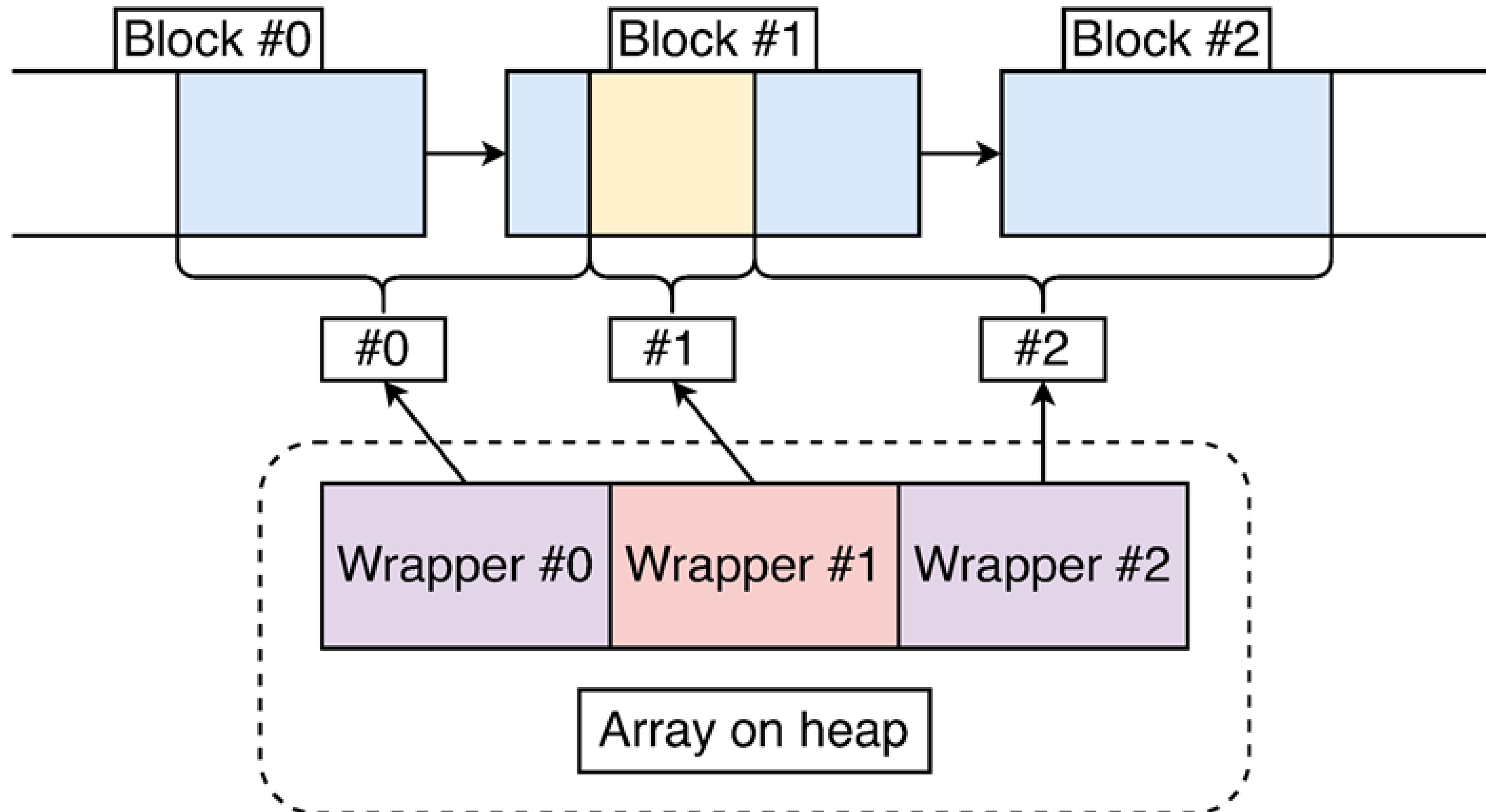
Реализация оптимизации

Чтение динамических объектов



Реализация оптимизации

Чтение массивов динамических объектов



Бенчмарк оптимизации

-10x

сократилось использование памяти в пике

+130%

парсинг стал быстрее

-40%

ухудшилась скорость доступа

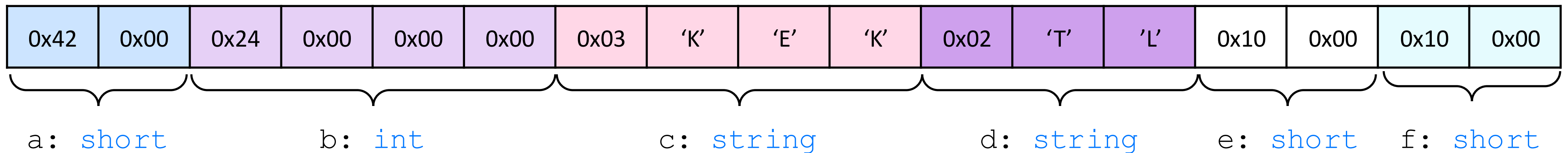
Если проходов по данным немного — суммарно выигрываем

Если скорость прохода данных мала в сравнении с временем доступа к диску — проигрыш по времени не чувствуется

Несовместимые изменения в формате

- Изначально у нас есть байтовая строка
- Непонятно, где какое поле начинается

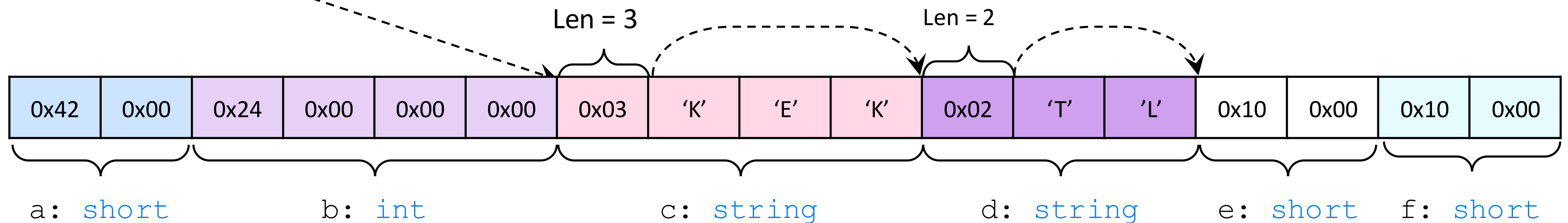
```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```



Несовместимые изменения в формате

- Проходимся первый раз по байтовой строке, размечаем начала объектов
- Ничего не копируем, строки просто пропускаем

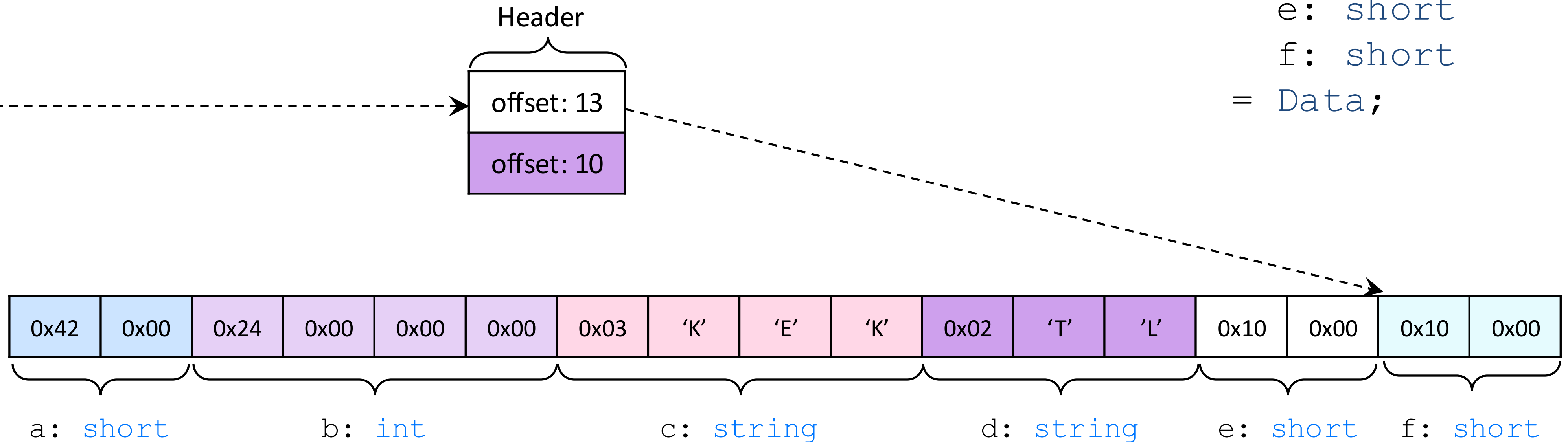
```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```



Несовместимые изменения в формате

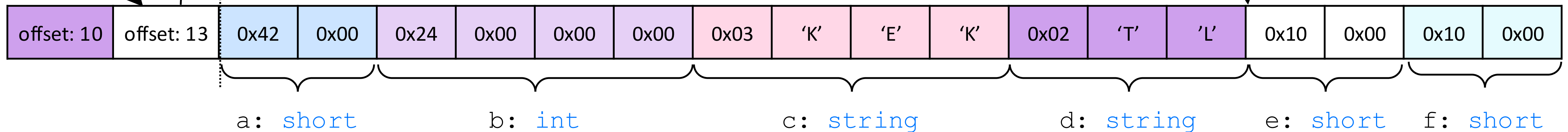
- Получаем заголовок, с помощью которого можно за O(1) обращаться к полям объекта

```
data
   a: short
   b: int
   c: string
   d: string
   e: short
   f: short
= Data;
```



Несовместимые изменения в формате

- Можно сохранять компактный заголовок объекта до начала самого объекта
- Тогда можно начинать обращаться к полям без необходимости обходить всю байтовую строку



Thanks for your attention

my dudes