Distributed S-Net: design and implementation

Grelck, C.; Julku, J.; Penczek, F.

Publication date
2009

Published in
Draft proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (IFL 2009)

Distributed S-Net Design and Implementation*

Clemens Grelck[1,2], Jukka Julku[3,4], and Frank Penczek[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl
[2] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{f.penczek,c.grelck}@herts.ac.uk
[3] Helsinki University of Technology
Otakaari 1, FI-02150 Espoo, Finland
[4] VTT Technical Research Center of Finland
Espoo, Finland
jukka.julku@vtt.fi

Abstract. S-Net is a declarative coordination language and component technology aimed at modern multi-core/many-core architectures and systems-on-chip. It builds on the concept of stream processing to structure networks of communicating asynchronous components, which can be implemented using a conventional (sequential) language.
In this paper we present Distributed S-Net, a conservative language extension for placement of components and component networks in distributed memory environments from compute clusters to wide-area grids. We further describe a novel distributed runtime system layer that complements the existing multithreaded runtime system for smaller shared memory multiprocessor and multicore machines. Particular emphasis is put on efficient management of data communication. Last not least, we present very preliminary experimental data.

1 Introduction

S-Net [1] is a novel coordination language and component technology that aims at facilitating the parallelisation of both existing and new applications. The design of S-Net is built on separation of concerns as the key design principle: an *application engineer* uses domain-specific knowledge to provide application building blocks of suitable granularity in the form of (rather conventional) functions that map inputs into outputs. In a complementary way, a *concurrency engineer* uses his expert knowledge on target architectures and concurrency in general to orchestrate the (sequential) building blocks into a parallel application.

In fact, S-Net turns regular functions/procedures implemented in a conventional language into asynchronous, state-less components communicating via

* This work was funded by the European Union Æther project grant.

uni-directional streams. The choice of a component language solely depend on the application domain of the components itself. In principle, any conventional programming language can be used, and a single S-Net network can manage components implemented using different languages.[5] Fig. 1 shows an example of an S-Net streaming network.

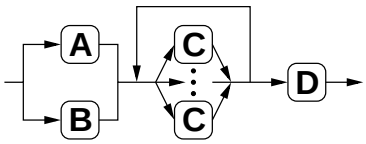

Fig. 1: Illustration of an S-Net streaming network of asynchronous components

Note that any base component is characterised by a single input and a single output stream. This restriction is motivated, again, by the principle of separation of concerns: The concern of a box is mapping input values into output values, whereas its purpose within a streaming network is entirely opaque to the box itself. Concurrency concerns like synchronisation and routing that immediately become evident if a box had multiple input streams or multiple output streams, respectively, are kept away from boxes. Our solution achieves a near-complete separation of computing and coordination aspects. We have identified four fundamental construction principles for streaming networks:

- *serial composition* of two (potentially) different components where the output stream of one component becomes the input stream of the other;
- *parallel composition* of two (potentially) different networks where some routing oracle decides on which branch data takes;
- *serial replication* of a single network where data is streamed through the same network a dynamically determined number of times; and
- *indexed parallel replication* of a single network where an index attached to the data determines which branch (or which replica of the network) is taken.

These four construction principles allow concurrency engineers to define complex streaming networks of asynchronous components and to turn sequential code blocks into a parallel application.

As high-level coordination language, S-Net in general is not bound to any memory model. The language concepts, however, fit in rather well with the basic concept of programming distributed-memory systems, i.e. message passing. S-Net boxes and networks are indeed asynchronous components that communicate with each other by sending messages via communication channels. In principle, the language could be used to define distributed memory systems as

[5] There are, however, technical limitations on the interoperability of languages and on the technical interplay between coordination and computation layer.

it is by mapping components directly to nodes of the system. However, direct mapping of components may not be sensible as we must take the cost of data transfers between nodes into account. Execution times of components may vary significantly from simple filters performing lightweight operations to boxes consisting of heavy computations. Another obstacle is the dynamic nature of S-Net networks that evolve over time due to serial and parallel replication.

What we need instead of a one-to-one mapping of boxes to compute nodes is a veritable distribution layer within an S-Net network where coarse-grained network *islands* are mapped to different compute nodes while within each such node networks excute using the existing shared memory multithreaded runtime system [2]. Each of these islands consists of a number of not necessarily continuous networks of components that interact via shared-memory internally. Only S-Net streams that connect components on different nodes are implemented by means of message passing. From the programmer's perspective, however, the implementation of individual streams on the language level by either shared memory buffers or distributed memory message passing is entirely transparent.

In principle, it would be desirable if the decomposition of networks into islands would be transparent as well, thus resulting in a fully implicit parallelisation architecture, that balances itself autonomously as the network evolves over time. With our shared memory runtime system, we have done exactly this [2]. However, given the substantial cost of inter-node data communication in relation to intra-node communication between S-Net components the right selection of islands is crucial to the overall runtime performance of a network. Therefore, we postponed the idea of a autonomously dynamically self balancing distributed memory runtime system for now and instead carefully extend the language in order to give the programmer control over placement of boxes and networks. In addition to the four above mentioned construction principles of networks we add two more:

- static placement of a network on some node;
- indexed placement of a network where an index attached to the data determines the node on which that data is to be routed to.

. These extensions are transparent with respect to S-Net semantics.

We chose MPI [3] as the middleware for our implementation mainly for its paramount availability and efficiency of implementation. Hence, node identification is by simple integer numbers, and the mapping of numbers to concrete machines in a cluster or wide-area network is beyond the scope of this work.

The specific contributions of the paper are

- the proposal of a conservative language extension for semi-explicit placement of networks;
- description of a distributed memory runtime system implementation on top of the existing multithreaded runtime system of S-Net;
- outline of a data manager service for optimised communication;
- preliminary performance figures.

The remainder of the paper is organised as follows: In Section 2 we provide a more detailed introduction to S-Net, while Section 3 introduces a running example that is used throughout the remainder of the paper. Section 4 describes the language extensions of Distributed S-Net in greater detail. Sections 5 and 6 illustrate the distributed runtime system and the design of the data manager, respectively. Eventually, we provide some preliminary runtime figures in Section 7 and conclude in Section 8.

2 S-Net in a nutshell

As a pure coordination language S-Net relies on a separate component language to describe computations. Such components are named *boxes* in S-Net terminology, their implementation language *box language*. Any box is connected to the rest of the network by two typed streams: an input stream and an output stream. Messages on these typed streams are organised as non-recursive records, i.e. label-value pairs. Labels are subdivided into *fields* and *tags*. Fields are associated with values from the box language domain. They are entirely opaque to S-Net. Tags are associated with integer numbers that are accessible both on the S-Net and the box language level. Tag labels are distinguished from field labels by angular brackets.

On the S-Net level, the behaviour of a box is declared by a *type signature*: a mapping from an *input type* to a disjunction of *output types*. For example,

```
box foo ({a,<b>} -> {c} | {c,d,<e>})
```

declares a box that expects records with a field labelled `a` and a tag labelled `b`. The box responds with a number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. Both the number of output records and the choice of variants are at the discretion of the box implementation alone.

As soon as a record is available on the input stream, a box consumes that record, applies its box function to the record and emits the resulting records on its output stream. In the simple but common case of a one-to-one mapping between input and output records the box function's result value may determine the output record. In the general case, our *box language interface* provides a box language specific abstraction named `snet_out` to dynamically produce output records during the execution of the box function. As soon as the evaluation of the box function is complete, the S-Net box is ready to receive and process the next input record.

S-Net boxes are stateless by definition, i.e., the mapping of an input record to a stream of output records is free of side-effects or, in other words, purely functional. We exploit this property for cheap relocation and re-instantiation of boxes; it distinguishes S-Net from conventional component technologies. In particular if boxes are implemented using imperative languages, S-Net, however, can only guarantee that box functions actually adhere to the *box language contract* as far as the box language supports such guarantees. This is in the end the same in any functional language that supports calling non-functional code.

In fact, the above type signature makes box `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends nicely to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

Subtyping on the input type of a box means that a box may receive input records that contain more fields and tags than the box is supposed to process. Such fields and tags are retrieved from the record before the box starts processing and are added to each record emitted by the box in response to this input record, unless the output record already contains a field or tag of the same name. We call this behaviour *flow inheritance*. In conjunction, record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were developed in isolation to cooperate with each other in a streaming network.

It is a distinguishing feature of S-Net that we do not explicitly introduce streams as objects. Instead, we use algebraic formulae to define the connectivity of boxes. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. As pointed out earlier, S-Net supports four network construction principles: static serial/parallel composition and dynamic serial/-parallel replication. We build S-Net on these construction principles because they are pairwise orthogonal, each represents a fundamental principle of composition beyond the concrete application to streaming networks (i.e. serialisation, branching, recursion, indexing), they naturally express the prevailing models of parallelism (i.e. task parallelism, pipeline parallelism, data parallelism) and, last not least, we believe that these four principles are sufficient to construct all useful streaming networks. The four network construction principles are embodied by *network combinators*. They all preserve the SISO property: any network, regardless of its complexity, again is a SISO component.

Let `A` and `B` denote two S-Net networks or boxes. Serial composition (denoted `A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B` while the input stream of `A` and the output stream of `B` become the input and output streams of the compound network, respectively. As a consequence, instances of `A` and `B` operate asynchronously is a pipelined fashion. In the intuitive example of Fig. 1 serial composition can be identified between the left, the middle and the right subnetworks.

Parallel composition (denoted `(A|B)`) constructs a network where all incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the compound network. By means of type inference [4] we associate each operand network with a type signature similar to the annotated type signatures of boxes. Any incoming record is directed towards the operand network whose input type better matches the type of the record itself. If both branches in the streaming network match equally well, one is selected non-deterministically. The example network in Fig. 1 features parallel composition in combining `A` and `B`.

Serial replication (denoted `A*type`) constructs a conceptually infinite chain of instances of `A`. The chain is tapped before every instance to extract records that match the type pattern given as right operand (i.e. the record's type is a subtype of specified type). Such records are merged into the output stream. In a simplifying view Fig. 1 illustrates serial replication as a feedback loop. While in a completely stateless setting feedback and replication are equivalent, the presence of synchronisation facilities (see below) requires us to make this subtle difference. From a conceptual point of view, their relationship resembles that of recursion and iteration; from a pragmatic point of view, the separation of data in different instances of the operand network contributes to an orderly system behaviour.

Indexed parallel replication (denoted `A!<tag>`) replicates instances of `A` in parallel. Unlike in static parallel composition we do base routing on types and the best-match rule, but on a tag specified as right operand. All incoming records must feature this tag; its value determines the instance of the left operand the record is sent to. Output records are non-deterministically merged into a single output stream similar to parallel composition. In Fig. 1 we can identify parallel replication of network `C`. To summarise we can express the S-Net sketched out in Fig. 1 by the following expression:

```
(A|B) .. (C!<t>)*{p} .. D
```

assuming previous definitions of `A`, `B`, `C` and `D`. While this example remains in the abstract, concrete S-Net applications can be found in [5, 6].

Last not least, S-Net features a synchronisation component that we call *synchrocell*; it takes the syntactic form `[|type,type|]`. Similar to serial replication the types act as patterns for incoming records. A record that matches one of the patterns is kept in the synchrocell. As soon as a record arrives that matches the other pattern, the two records are merged into one, which is forwarded to the output stream. Incoming records that only match previously matched patterns are immediately forwarded to the output stream. Hence, a synchrocell becomes an identity after successful synchronisation and may be removed by a runtime system. The extremely simplified behaviour of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continuous synchronisation of matching pairs in the input stream, can easily be achieved using synchrocells and network combinators. See [5] for more details on this and on the S-Net language in general.

3 Running Example

Our running example is a very simple dictionary-based password cracker. It takes a dictionary and number of Md5-encoded passwords as its input and produces the corresponding decoded password for each entry that can be cracked with the given dictionary. The cracking is done by encrypting words of the dictionary one by one and comparing the resulting hash value with the encoded password. Each password is associated with a cryptographic salt to make the cracking more time-consuming. Fig. 2 shows the S-Net implementation.

```
net crypto ({dict, entries, <dict_size>, <num_entries>, <num_branches>}
              -> {word, <entry>} | {<false>, <entry>})
{
   box splitter ({entries, <num_entries>}
                   -> {password, salt, <entry>});

   box cracker ({password, salt, dict, <dict_size>}
                  -> {word} | {<false>});

   net load_balancer
   connect [{<entry>, <num_branches>}
              -> {<entry>, <branch = entry % num_branches>}];
}
connect splitter .. load_balancer .. cracker!<branch>;
```

Fig. 2: S-Net code of running example: password cracker

The code defines a network named `crypto` that consumes records containing two fields and three tags. The field `dictionary` contains the dictionary and the field `entries` contains a list of all the passwords and their salts. The tags `dictionary_size` and `num_entries` contain the number of words in the dictionary and the number of passwords, respectively. The tag `num_branches` is used to define in how many parallel branches the processing can be made. The network produces records that either contain the decoded word and the number of the password or a tag that indicates that the password could not be cracked.

The crypto network consists of two boxes and one subnetwork. The box `splitter` takes records that hold the field containing the passwords and their salts and the tag representing the number of passwords and splits these records into smaller records, each holding fields for one password and its salt and a tag containing the ordinal number of the password. The box `cracker` does the actual password cracking. It consumes records containing the password data and the dictionary and produces decoded words or `false` tags in case the password could not be cracked. The subnetwork `load_balancer` consists of a single filter that takes records containing the ordinal number produced by the `splitter` box and assigns each record a branch number according to the ordinal number, forming a simple round-robin scheduler together with the index split combinator around the `cracker` box.

The records flowing in the `crypto` network are first passed in the box `splitter` which is then serially connected to the `load_balancer` network. This combination is then serially connected to the next network which is built by embedding the box `cracker` into an index split combination. The index split combinator is controlled by the tag `branch` assigned by the `load_balancer`, which means that the work is shared between `num_branches` parallel `cracker` boxes. This allows the time-consuming decoding operation to be performed in parallel to multiple passwords in case the system contains more than one processing unit. The reader should note that because of S-Net's flow-inheritance properties all the fields and tags that are not mentioned in the network and box signatures are automatically inherited to each record that is produced as the result of the record containing the inherited data.

4 Distributed S-Net

We extend S-Net by two placement combinators that allow the programmer to map networks to processing nodes either statically or dynamically based on the value of a tag contained in the data. Let `A` denote an S-Net network or box. Static placement (written `A@42`) maps the given network or box statically to one node, here node 42. Location assigned to a network recursively applies to all of those subnetworks and boxes within the network whose location is not explicitly specified by another placement combinator. If no location is specified at the outermost scope of S-Net network definition hierarchy, a default location, zero, is used instead.

The second placement combinator is actually an extension of the indexed parallel replication combinator. Instead of building multiple local instances of the argument network, it distributes those instances over several nodes. Let `A` denote an S-Net network or box, then `A!@<`*tag*`>` creates instances of `A` on each node referred to by `<`*tag*`>` in a demand driven way. Effectively, this combinator behaves very much like regular indexed parallel replication, the only difference being that each instance of `A` is located on a different node.

Placement combinators split a network into sections that are located at the same node. Each of the logical nodes may contain any number of these sections. Sections located in the same node are executed in the same shared memory, which means that data produced on one section can be consumed in another section on the same node without any data transfers between address spaces.

We use ordinal numbers as the least coomon denominator to identify nodes. These nodes are purely logical: nodes define parts of S-Net stream that are located in the same shared-memory. Language notation itself doesn't consider how the nodes are mapped to the actual physical nodes, but the mapping from the logical nodes to the physical nodes is S-Net implementation specific. The motivation for this is that defining the actual physical nodes in the language level would bind the program to the exact system defined at compile time. Using logical nodes allows the decisions about the physical distribution to be made at startup time. With MPI as our current middleware of choice the number directly reflects an MPI node. In more grid-like environments it may be more desirable to have a URL instead. We consider this mapping of numbers to actual nodes to be beyond the scope of S-Net.

When the placement of a network is defined, the end of the input and the beginning of the output stream of the network are always located on the given node. If the location of the network is not explicitly defined, the end of the input stream of the network is located in the node where the first component of the network is located at. Correspondingly, the beginning of the output stream of the network is located on the node in which the last component of the network is located at. The input and output streams of a network do not have to be on the same node. This feature allows an S-Net application to move data from one node to another while processing it.

Fig. 3 shows a distributed version of our running example introduced in the previous section; a graphical representation of the network can be found in Fig. 4.

```
net crypto ({dict, entries, <dict_size>, <num_entries>,
             <num_nodes>, <num_branches>}
                -> {word, <entry>} | {<false>, <entry>})
{
   box splitter ({entries, <num_entries>}
                   -> {password, salt, <entry>});

   net load_balancer ({<entry>, <num_nodes>, <num_branches>}
                        -> {<entry>, <node>, <branch>})
   connect [{<entry>, <num_nodes>, <num_branches>}
              -> {<entry>, <node = entry % num_nodes>,
                  <branch = (entry / num_nodes) % num_branches>}];

   net divider ({password, salt, dict, <dict_size>, <branch>}
                   -> {word} | {<false>})
   {
      box cracker ((password, salt, dict, <dict_size>)
                     -> (word) | (<false>));
   }
   connect cracker!<branch>;
}
connect splitter .. load_balancer .. divider !@ <node>;
```

Fig. 3: Distributed S-Net specification of running example

We assume a system that consists of multiple computing nodes each of which contains a number of processing units, i.e. processors or cores. If each node had only contained a single processor, it would be straightforward to run a single `cracker` box on each node, and replacing the original index split combinator around the box `cracker` by a placement split combinator would have achieved exactly this. However, assuming nodes with more processing power, we have wrapped the box `cracker` and the index split combinator inside another subnetwork that is embedded into a placement split combinator. This solution with the help of information about the number of nodes and the updated `load_balancer` network extend the record scheduling scheme to manage multiple nodes each containing the same number of boxes. As the result of these modifications the S-Net network is spread over multiple computing nodes. The initialization tasks including the splitting of the data and the load balancer are still executed on the same node. The new `divider` subnetwork will be built into each of the nodes and the records are scheduled to each instance of the network in round-robin fashion.

5 Towards a Distributed Runtime System

The runtime support for distributed-memory systems is built as a separate layer on top of our existing shared memory runtime system. One of the main design-principles is to separate these layers as completely from each other as possible. In principle, no S-Net component needs to know about the distribution as the distribution layer is entirely hidden by the realisation of streams. This design facilitates maintenance and further development of both the shared and the distributed memory versions inside the same code base.

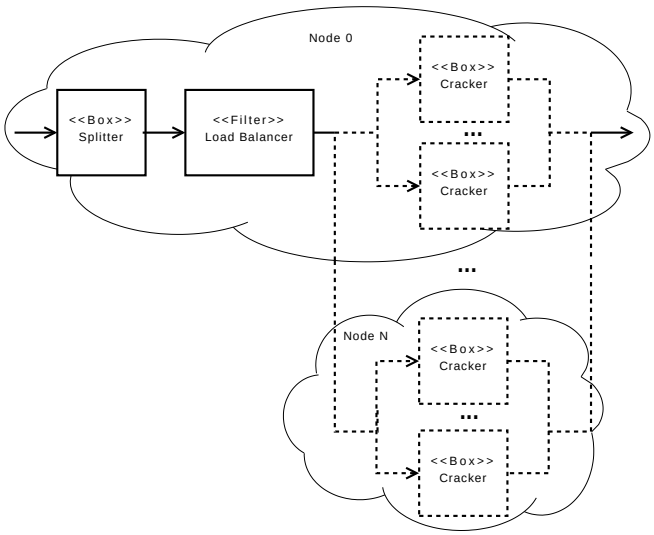

Fig. 4: Illustration of the network presented in Fig. 3

As mentioned before we chose MPI as middleware for its wide-spread availability and because it satisfies our basic needs for asynchronous point-to-point communication and data marshalling. Each of the logical nodes is implemented as an MPI process. The logical node identifiers defined at the S-Net language level correspond directly to MPI process ranks. Accordingly, we leave the exact mapping of logical nodes to physical resources to the MPI implementation.

To provide scalability to S-Net runtime system implementation, the system nodes cooperate as peers: there is no central control or name servers in the system that could become a performance bottleneck. Each node is identical apart from the S-Net components it contains.

In the language level placement can be applied to any valid networks and boxes. The placement combinators divide the network representation into multiple sections, each containing continuous sequence of runtime components that are mapped into the same node. Each node may contain an unbounded number of sections like this. If a subnetwork of some network is mapped into a different node, a section is divided into multiple smaller sections. Due to parallel composition, each section may have more than one input and output stream.

The components do not send records directly to other nodes, but the boundaries between the nodes are hidden behind streams. To manage these streams each node has two active components: an input manager and an output manager. Figure 5 illustrates the architecture of a single node.

The output buffer of a section and the input buffer of the next section can be considered as instances of the same buffer on different nodes. Output and input managers transparently move records between these buffers. Both the managers

are implemented with multiple threads, one for each connection. The reason for this is that with blocking communication the threads can be used to propagate congestion of the streams to preceding nodes without blocking the whole node. Secondly, multi-threading is required to prevent dead-locks that could appear in single-threaded implementation in cases where the same stream goes through a node more than once. All the threads work completely independently and there is no shared state between them.

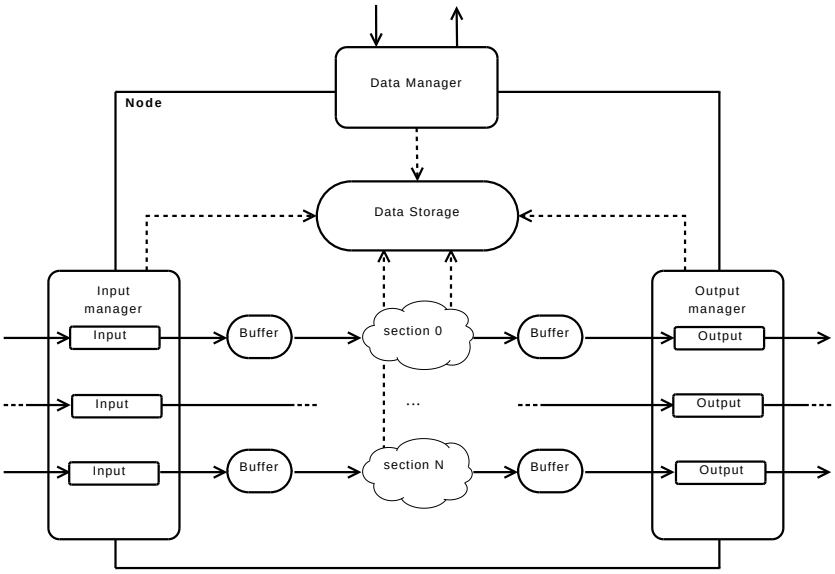

Fig. 5: Internal organisation of one node

The input manager consist of one *control thread* that listens for control messages sent by the other nodes and one *input thread* per stream that arrives in to the node. The control thread listens to requests to create new network sections and update messages that contain information about new connections. Update messages trigger creation of a new input thread. Each input thread listens to exactly one connection, deserialises incoming records and passes them to the input stream of the corresponding network section.

The output manager consists of an *output thread* per stream that leaves the node. Each thread serves as a counterpart for an input thread on some other node. Output threads simply serialise records and send them to the node containing the next section.

Data management is separated from the stream management. The box language data is not transferred between the nodes with the records. Instead only a representation of the data is sent and the real data is later fetched separately on

demand. The motivation for this is that a record may flow through several nodes before a particular data element is consumed. By fetching a data element only into those nodes where it is actually needed, unnecessary data transfers can be avoided. Another motivation to separate the data and the records is that, even though both of them are moved from one node to another, the needs may be quite different. For example, records are assumed to be relatively small messages, while the size of the data elements may range from bytes to gigabytes.

Another active component, the *data manager*, is used to handle data management needs of the distributed S-Net. References to all data elements are stored into data storage, a hash table that allows tracking of data elements currently residing on a node.

In general, the input manager controls all the communication between the nodes, except the communication related to remote data operations explained more in Section 6. This also gives the input manager an important role in creation of new network sections. This is discussed more in the next section.

6 Managing Data Communication

In distributed S-Net boxes are mapped to certain nodes. Hence, data needs to be moved between the nodes. Data transfers may have serious effect on the program performance, depending on the underlying system, amount of data to be moved and data access patterns of the program. The programmer has the main responsibility in achieving performance and can affect the performance by choosing the right data access patterns, that is, minimizing data transfers between the nodes. This section describe how the prototype implementation manages the box language data.

6.1 The approach

One of the S-Net's main design principles is to separate coordination from computation. As the result, the current shared-memory implementation of S-Net is almost unaware of the box language data. The data elements are simply collected into records and managed through opaque pointers and copy and delete functions offered by the language interfaces. Only the boxes know the data representation and can operate directly on the data. In distributed-memory the data management becomes somewhat more complex. To be able to move data between the nodes in potentially heterogeneous environment the runtime system requires more information about the data representation.

Motivated by the performance penalty of the data transfers, the main principle of the data management is to avoid any unnecessary data movement. In the shared-memory S-Net implementation each data element is directly stored into a record. In distributed S-Net, the box language data is not transferred inside the records over node boundaries, but instead only a representation of the data consisting of the labels and locations of each data element is included. The motivation for this is that the fact that a record is passed to a node does not

imply that all the data of the record is needed there. In fact, a record may flow through several nodes carrying exactly the same data, in which case most of the data transfers would be unnecessary. The tag values are always transferred with the records as they might be needed for routing purposes.

S-Net requires abstract copy and delete operations from the language interfaces, but doesn't particularly define how the operations are to be performed. The current language interface implementations for C and SAC use reference counting, which goes well hand-in-hand with S-Net's functional behaviour requirements for the boxes and is cheaper than deep copy operations. In the distributed-memory implementation, deep memory copies may be wasted in case the data is not used on the same node but instead transferred to another node after the copy. Reference counting is a cheaper operation in this case. On the other hand, always assuming reference counting unnecessarily limits the language interface implementation.

This problem is solved by postponing the language interface level copy until the copied data is actually needed. This is done by implementing reference counting mechanism inside the runtime system. The opaque data pointers in records are replaced by reference objects that are used to hide the reference counting and distribution of the data from the rest of the runtime system. This decision does not restrict possible future language interfaces that perform real copying instead of reference counting or inherent reference counting of any box language, as the corresponding language interface functions are still called if there are multiple instances of the data alive when some instance of the data is consumed.

There is also another catch in introducing the runtime system level reference counting. In addition to avoiding unnecessary copies, also unnecessary data transfers may be avoided in some cases. Because only the boxes can modify data and because of their functional behaviour, it is safe to make assumptions about which data elements are identical. Copies of a data element can be tracked and in case a node contains a real copy of the data at the time when another fetch for the same data occurs, the fetch can be satisfied much more cheaply by using the local copy instead and just adjusting the reference counts properly.

In a way, the memory management system of the prototype implementation can be thought as a very simple user-level software COMA [8] system where the data elements are freely replicated and migrated into local memory of any of the system nodes.

6.2 Naming of the data

Separation of the data from the records sets a new requirement for the runtime system: it must be possible to refer to any data element, local or remote, from any node. In a distributed memory environment the runtime system must be able to name all the data elements. Therefore, we introduce *unique data identifiers*, called UDIs. When a new data element is created a UDI is assigned to it. The identifiers need to be unique over space and time. The identifiers assigned by a node must also be different from all the identifiers assigned by all the other nodes; identifiers cannot be re-used as some other node might still refer to an older data

element with the same identifier [9]. It is generally undecidable whether or not some identifier is in use without synchronisation over all the nodes.

The uniqueness of the identifiers can be achieved by concatenating the node identifier to some locally unique identifier [9]. An UDI consists of the identifier of the node, that is, the MPI rank of the node, where the data was originally created and an integer identifier that is locally unique in that node. As the data element may move, using the address of the data as the unique part of the name is not doable. For this reason the locally unique part of the name consists of integer numbers that are allocated in increasing order to guarantee the uniqueness. For the exactly same reason, it is not sensible to use the name directly to locate a data element [9]. The UDI of the data does not change when the data is transferred to another node, but the location of the data is tracked separately with accuracy of a node.

UDIs also form the basis for reasoning which references point to the same data. All references to instances of the same data element are assigned the same UDI. Reasoning about the identical data elements is conservative. A data element that comes out of a box may be exactly the same that went in. However, as the runtime system lacks information about the box, it has to make a safe assumption and consider the data as a new element and provide it with a new UDI. When records flow through a network and across nodes, it is enough to keep note of the location of the last known real copy of each data element in the record. This is possible because copies of the same data located on different nodes are identical.

6.3 Data operations

There are basically two approaches to communicate data between processes: pushing or pulling. Pushing means that a process requests another process to take a data item. Pulling means that when a process needs a data element it asks it from another process. Pushing has the obvious benefit over pulling that it can be made in advance, which may remove or shorten the data transfer delay compared data fetch on demand.

In S-Net it is not generally possible to know where some data element is next needed. This makes sensible pushing of data to other nodes nearly impossible. However, because each box specifies which data it uses, the runtime system can transparently fetch data elements into correct nodes on demand. This implies a need for a data fetch operation. S-Net filters may copy data elements without need to use the data immediately or the data may be discarded without being used at all. This gives motivation for remote copy and delete operations: there is no need to move the data to another node for the operation, but the operation can as easily, but with cheaper communication, be performed on the node where the data currently resides.

Each node contains an active component called data manager that serves remote data operations. With help of the UDIs the data manager maps remote copy and delete operations to corresponding local operations and moves data elements to other nodes on demand. Data manager's implementation allows for several

concurrent fetches to be performed at the same time by using asynchronous communication. A unique identifier similar to UDIs is used to distinguish between the concurrent operations.

A data fetch operation consists of three messages. A *request message* identifying the data element is sent from the node where a data element is needed to the data manager of the node that currently holds the data. This message is answered by the data manager with a *data type message*. MPI requires that both the sender and the receiver know the type of the data for data marshalling purposes. The data type message contains serialised representation of the data type that is used to construct the corresponding MPI type on the fetching node. After both the nodes know the data type it is possible to transfer the actual data between the nodes. Both the remote copy and delete operations consist of a single message that identifies the corresponding data element and the operation to perform.

6.4 Refering the data

Each node contains an object called *reference* for each data element, local or remote, that can currently be referred within the node and for each local data element that can be remotely referred from any of the other nodes. A reference works like a proxy object [9]: it hides the actual location of the data from rest of the runtime system by forwarding the copy and the delete operations of remote data to nodes where the data is currently located at and by automatically fetching the data to the node when it is consumed.

A reference object contains the unique data identifier and the location of the data it is referring to and the count of references to the data through *that* particular reference. These references may be local or remote. If the real data is located at the node, the reference contains a pointer to it. References are stored into a hash table called *data storage* that allows fast finding of a specific reference.

In every record each data field points to some reference. If the actual data is needed on the node and is not yet located there, a remote fetch is triggered. Concurrent fetches of the same data from the same node are prevented. After a fetch completes all the records pointing to the same reference can automatically use the same fetched copy of the data. The data is further copied with local language interface copy function if needed. When a record leaves a node that does not contain an instance of the actual data, the location of that data in the record is changed to refer the node where the real data resides. This is done to prevent references from forming chains that would have to be collapsed later. Direct references to the data also remove the need for name servers or others comparable systems that could become an obstacle to scalability. For simplicity reasons, a local reference may not point to more than one remote reference at the time. If a record ever arrives to a node that already contains a reference, local or remote, to a data item that the record refers to, the record is changed to use that reference instead of the one it carries.

Fig. 6 illustrates data management by means of an example. the original instance of the data resides on node *one* and is locally referred by a single record. A reference to the data has been carried by a record to every other node in the system. The data has not yet been consumed on nodes *two* and *four*, but the data has been copied there. As the result, both the nodes contain several records referring to the same data. The local reference in these nodes points to the original reference in node *one*. In node *three* the case is somewhat different: there have been at least two references to the data, and one of them has been consumed. Because of this, the real data has been fetched into the node and all the local references point to the local copy instead of the original copy on node *one*. In case the record on node *three* is moved, for example to node *four*, the record is modified to point the reference on that node and the data in the node *three* is deleted, as the last reference to it would be lost.

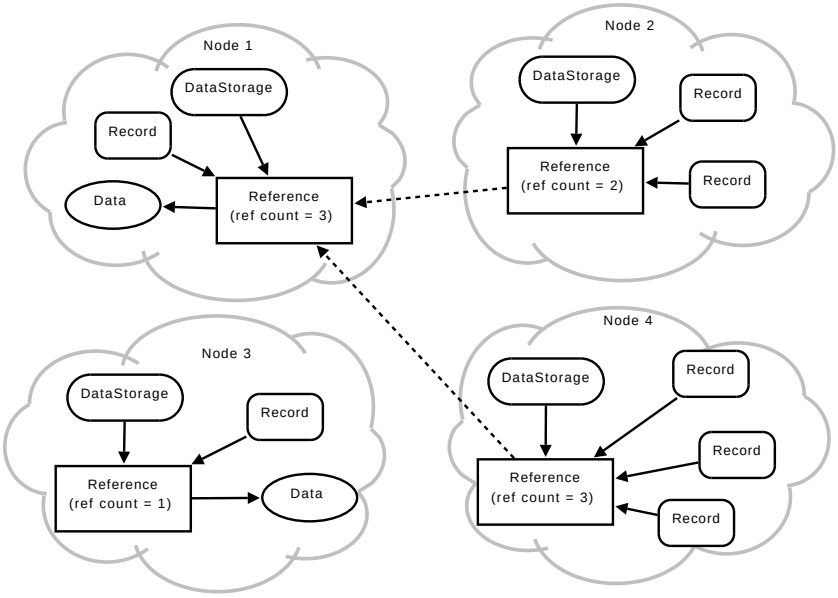

Fig. 6: An example of data references

7 Experimental Evaluation

We experimentally evaluate our approach using the running example of a password cracker. Our test environment is a small cluster of 1.8 GHz Dual-Core AMD Opteron 1210 processors running Ubuntu Linux 2.6.24 and MPICH-2 1.1a2 [10] with maximum threading support enabled. The nodes are connected via gigabit

Ethernet and use SSH communications. The MPI and S-Net libraries and the program code are shared by the nodes through NFS. The boxes of the program are implemented as C language functions using S-Net's C language interface. We use Ubuntu's British-English dictionary and three data set sizes of 12, 36 and 60 entries. The number of entries in data set was chosen so that the load in each processor core would be approximately the same. The same encrypted word was used for each entry to produce more even loads. The middle word of the dictionary was chosen for this.

nodes	12	36	60
1	1.006	1.002	1.002
2	0.503	0.502	0.504
3	0.337	0.337	0.337

Fig. 7: Relative all clock execution times of running example using 1, 2 or 3 nodes and dictionaries with 12, 36 and 60 entries.

Fig. 7 shows relative wall clock execution times measured on our experimental setup, i.e. the execution time of a distributed version of our running example on 1, 2 or 3 nodes divided by the execution time of the non-distributed version on a single node. We observe nearly linear speedups for all size classes. This may be less a surprise as the application is indeed embarrassingly parallel. However, one needs to take into account that these speedups on a distributed memory architecture have been achieved without any classical parallel programming involved, solely by means of S-Net coordination of conventional C-implemented components. Unfortunately, we have not had access to a larger computing cluster that would have allowed for further investigations into scalability.

8 Conclusion

The S-Net language was extended to include two new network combinators: a static placement combinator and an indexed dynamic placement combinator. They allow programmers to partition an S-Net network over several compute nodes. As a result the runtime system deals with two levels of concurrency: coarse-grained concurrency on the level of compute nodes using distributed memory communication and fine-grained concurrency within each node using shared memory communication managed by our existing runtime system [2].

The main challenges addressed by the implementation are the dynamic construction of the S-Net network runtime representation spanning over several nodes, routing of records between the nodes and data management problems caused by the separation of the network into multiple distinct address spaces. Very preliminary experiments show that the approach taken allows us to achieve good speedups on distributed memory clusters without compromising the high-level programming style of S-Net.

The most obvious future work is to gather more experience with the current system using real-world applications and larger clusters. Various aspects of our distributed runtime system need further improvement. Examples for these are prefetching of data from remote nodes before a box actually starts processing data or the collection of several small messages into a single larger one to reduce message passing overhead.

An interesting area of future research is the combination of Distributed S-Net as described here with the ongoing research on reconfiguration and self-adaptivity in S-Net [11]. In conjunction, the two lines of research add further expressiveness to S-Net: distributions of networks across distributed memory environments can dynamically be changed either through external events (reconfiguration) or internal observation (self-adaptivity).

References

1. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237
2. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S.B., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
3. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, Cambridge, Massachusetts, USA (1994)
4. Cai, H., Eisenbach, S., Grelck, C., Penczek, F., Scholz, S.B., Shafarenko, A.: S-Net Type System and Operational Semantics. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)
5. Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2009)
6. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)
7. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.1. (2008) http://www.mpi-forum.org/docs/mpi21-report.pdf.
8. Protic, J., Tomasevic, M., Milutinovic, V.: Distributed Shared Memory: Concepts and Systems. John Wiley and Sons (1998)
9. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems Concepts and design. 4th edn. Addison-Wesley (2005)
10. Argonne National Laboratory: (MPICH2) http://www.mcs.anl.gov/mpi/mpich2.
11. Penczek, F., Scholz, S.B., Grelck, C.: Towards Reconfiguration and Self-Adaptivity in S-Net. In Scholz, S.B., ed.: Implementation and Application of Functional Languages, 20th international symposium, IFL'08, Hatfield, Hertfordshire, UK. Technical Report 474, University of Hertfordshire, England, UK (2008) 330–339