Regular Paper

Tuplink: A Meta-middleware System for Micro-clients

Yasushi Negishi,† Kiyokuni Kawachiya,† Hiroki Murata† and Kazuya Tago†

Progress in semiconductor technology has made it possible to build small network clients compact enough to be embedded in credit cards or wallets. These devices, which have severely restricted computing resources, are called "micro-clients." We propose an approach for building systems for micro-clients. The requirements of a system for micro-clients are as follows: (1) It must work with a small amount of memory and a low-power processor. (2) It must work even while the link is disconnected, because of the high cost of communication. (3) It must work with low-quality communication links. We introduce a software system called Tuplink that meets these requirements. The Tuplink system on the client node manages a central data pool to hold user and system information in an integrated manner. The server node also has a data pool for each client, and the contents of both pools are kept identical by means of a communication network. This server-side data pool makes it easier to build a system in which necessary functions are divided between server and client nodes. One-to-one communication between client and server is abstracted by a synchronization operation between the two pools in order to hide link management and communication timing from other subsystems. A communication protocol for synchronization, called the Tuplink protocol, is used to synchronize the two pools. Other subsystems, such as the RPC, file, and database systems, use the central data pool instead of their own buffers. The state of the communication link is hidden from other subsystems. We call this the "meta-middleware" approach. The Tuplink system meets the above requirements as follows: (1) Use of the central data pool eliminates duplication of buffers among subsystems and copying of data among buffers. (2) The system can operate while the link is disconnected by using the central data pool as a data cache. (3) The synchronization protocol efficiently handles packet loss by using the central data pool as a communication buffer. We have built systems based on the Tuplink model, and applications for them on several platforms, including a smart phone, Palm III, and Windows CE. This paper discusses the approach and findings of the system implementation.

1. Introduction

As the network bandwidth grows, server nodes are coming to occupy a larger part of the functionality of network computing, while client nodes are becoming thinner. This improvement in hardware technology is making it possible to use a wider variety of client device types, and is greatly increasing the potential of network computing.

For example, even single-chip computers with I/O interface, memory, and processor are becoming powerful enough to be used as network nodes, and the power consumption of such chips is becoming low enough for them to operate on dry batteries. Such devices will make it possible to design extremely compact mobile clients. The chips can be embedded into wristwatches, telephones, wallets, and other personal belongings. If they are cheap enough to be disposable, such devices may become much more numerous than PCs, and they will allow people to use network computing services far more easily than they can now.

Realizing such a change will require a new approach to the design of system software for client devices. The most important point is to reduce the implementation cost by finding a way to provide a wide variety of services with a simple software structure and limited computation resources. When a system is integrated into a single chip by using current technology, the memory size is limited to 2 MB [1]. We call a network client that has a GUI and operates with this level of computing resources a "micro-client."

Though the resources are restricted, many new features are needed to allow micro-clients to be used as mobile clients. Operations need to be supported whether the link is connected or not, because wireless communications are costly and frequently disabled by physical conditions on account of unpredictable timing. Smooth and transparent transitions between connected and disconnected modes are also important.

We reconsidered what kind of approach to the

† IBM Research, Tokyo Research Laboratory

design of system software was needed to meet the requirements of micro-client design. In our new design, all system data are kept in a single common data pool, and the pool is replicated on client and server. Use of such a pool totally eliminates data copying between system buffers and significantly simplifies the system structure. We call this the "meta middleware" approach, and we call the middleware system Tuplink. Systems on micro-clients, such as file, database, and RPC systems, are implemented on the basis of the meta-middleware.

In the rest of this paper, we point out the requirements of system software for micro-clients in Section 2, outline the proposed approach in Section 3, show the protocol for the synchronization in Section 4, describe the implementation of the proposed system in Section 5, discuss the effectiveness of the approach in Section 6, mention related work in Section 7, and present our conclusions in Section 8.

2. Requirements of System Software for Micro-clients

We call a network client that meets the following conditions a "micro-client."

(1) It is cheap and small, and can often be embedded in personal belongings, such as credit cards, wallets and watches.
(2) It uses a private or public wireless link.
(3) It has a graphic user interface for interacting with users.

System software for micro-clients has the following three requirements.

2.1 Limited Resource Support

System software for micro-clients must work with little memory and a low-power processor. Micro-clients, such as smart phones, wallets, and credit cards, must be cheap and must work with low-power batteries, and therefore they have severely restricted resources, such as a 16-bit processor and 100 KB of RAM.

2.2 Disconnected Mode Support

Micro-clients usually use a public wireless link, because they move in wide areas. Since the cost of a public wireless link is expensive, and depends on the length of time connected, the system must work even while the communication link is disconnected. A user operates a micro-client even while the link is disconnected, and connects the link only when necessary.

2.3 Low-Quality Link Support

Public wireless links have lower quality than LANs. The following types of support are necessary:

Sudden Disconnection. System software needs to hide from the application any failure of a communication link at any time. Existing system interfaces frequently lack support for this type of encapsulation.

Accidental Packet Loss. TCP/IP's congestion control mechanism is started by even a single packet loss, because almost all packet losses are caused by buffer overflow on gateway machines when LANs are used. However, it causes too much performance loss, because public wireless links have accidental packet losses. System software must work with accidental packet losses in communication links.

3. Tuplink Approach

To meet the requirements described in Section 2, we have developed a new communication/data management system named "Tuplink." In the Tuplink system, an integrated data pool is prepared in both server and client. In the data pool, all data are placed as immutable "tuples." The data pools in the server and client are synchronized by means of a communication link.

3.1 System Data Pool

It is frequently pointed out that there are many functional duplications among the components of today's system software [2),3)]. For example, the functional duplication between operating systems and database management systems has been discussed for decades. Another cause of inefficiency is the use of a number of buffers in various system layers. Distributed file systems are a typical example. The I/O system of communication link, protocol stack, RPC system, and file system uses different types of buffers in an ad hoc way in implementing a distributed file system. The use of multiple buffers leads to the consumption of excessive memory for the buffering area, and also to the consumption of excessive processing power owing to data copying between the buffers [3)]. This will be a serious concern as regards micro-client implementation. It is also important to reduce the processing power in the micro-client design so as to reduce the power consumption and thus allow the use of a smaller battery.

To solve these problems, we use a data pool with the following criteria.

(1) In each node, use a single data pool for all system and user data, to eliminate du-

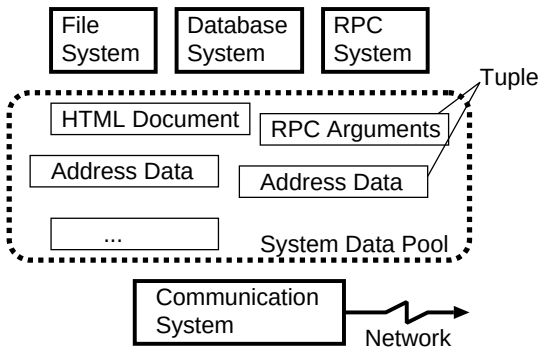

Fig. 1 Data pool and system components on a client node.

plicated buffers and data copying among them.

(2) Place copies of the data pool on both client and server nodes. Communication between client and server is abstracted by a synchronization operation between the two pools in order to hide link management and communication timing from the applications.

Figure 1 overviews the structure of the system software of a client node. The communication system places received data in the pool. An immutable unit of data in the pool is called a "tuple." A tuple keeps a set of triplets of tag name, data type, and value. Other system components, such as the database management system and RPC system, use the tuple, without copying or changing the data representation, for keeping all kinds of unit system information such as unit data of the database management system, queue elements, and RPC argument buffer. The data pool is called the "system data pool."

The idea of the system data pool comes from that of Linda Tuple Space, introduced by Gelernter [4]. Linda is a programming language for parallel/distributed computing. It uses a data pool called Tuple Space, which is shared by all the processors and machines. Our Tuplink uses concepts similar to Linda's tuple and Tuple Space, but its objective and mechanism are different.

3.2 Operations

Tuplink's system data pool is managed by the following operations. We summarize these operations here. Details of the operations are given in the following subsections.

The first three operations are used to manage each system data pool. They are "out" for creating a tuple, "read" for referencing a tuple, and "in" for retrieving a tuple.

out Create a tuple with the specified data structure.

read Refer to a tuple that matches the specified condition. Both blocking and non-blocking modes are supported.

in Retrieve and delete a tuple that matches the specified condition. Both blocking and non-blocking modes are supported.

A tuple in Tuplink is immutable like Linda, so there is no operation for modifying a tuple. Although the basic meanings of the operations of Tuplink are the same as those of Linda, the data structures of a tuple and a condition differ. A tuple in Tuplink consists of an unordered set of triples of "tag name," "data type," and "value." The structure of a condition for retrieving a tuple is the same as that of a tuple. A condition matches a tuple when the condition is a subset of the tuple. Details of the data structures and the matching mechanism are given in the following subsections.

The three operations introduce a basic communication concept called "Generative Communication," which is the same as that in Linda. See Gelernter [4] for the concept of Generative Communication.

On the other hand, the following three operations are not used in Linda, because Linda assumes permanent link connection. Tuplink introduces the following operations for managing link state.

sync Synchronize two corresponding system data pools. The link is connected before the synchronization and disconnected after it is finished.

connect Connect the link. Two system data pools are being synchronized while the link is connected.

disconnect Disconnect the link.

3.3 Data Structure of a Tuple

An "out" operation creates a tuple with the specified data structure. In Linda, a tuple consists of an ordered set of pairs of data type and value, such as

```
["Tiger Woods"(STR),
 24(NUM),
 false(BOOL),
 187(NUM)].
```

Whereas in Tuplink, a tuple consists of an unordered set of triplets, called an "Item," of tag name, data type, and value. The following is a sample of a tuple in Tuplink:

```
["Name" = "Tiger Woods" (STR),
 "Age" = 24 (NUM),
 "Smoker" = false (BOOL),
 "Height" = 187 (NUM)]
```

In this example, each line shows an item. The first column, such as "Name," shows the tag name of each item. The second column, such as "Tiger Woods" or false, shows a value. The third column, such as (STR) or (NUM), shows a data type. We will use this notation of tuple in this paper. The following five types of tuple value are supported.

NUL Special type denoting a null value
NUM Number
BOOL Logical (Boolean) value (true/false)
STR String
BYTE Byte sequence and length

The reason Tuplink uses unordered sets is because a tuple is shared by some system components, and therefore the data structure of a tuple should be extendable. In Tuplink, a program can use a tuple with a new item without loosing compatibility with the existing data structure.

3.4 Data Structure of Matching Conditions

A "read" operation refers to a tuple that matches the specified condition. An "in" operation retrieves and deletes a tuple that matches the specified condition.

In Tuplink, the data structure of matching conditions is the same as that of a tuple. A matching condition matches a tuple when the condition is a subset of the tuple. The following is a example of a matching pair of a condition and a tuple.

```
Matching Condition:
["Name" = "Tiger Woods"(STR),
 "Age" = 24(NUM)]
Tuple:
["Name" = "Tiger Woods"(STR),
 "Height" = 187(NUM),
 "Age" = 24(NUM),
 "Smoker" = false(BOOL)]
```

The data structure and matching scheme enable programs to use a tuple with new types of information independently of other programs. The Linda tuple representation depends on the number and order of items. Thus it is necessary to modify existing programs when another program uses a tuple with a different type of information from existing tuples.

3.5 Data Pool on the Server Side

In Tuplink, the server side also has a data pool to be synchronized with the data pool on the client side. This server-side data pool has the following merits:

(1) The server-side pool helps to build a client-server-type system. Because of the poor resources of micro-clients, functions on the client side should be minimized. In Tuplink, system components, such as database system and file system, are implemented as two separate modules, one on the client side and one on the server side.
(2) The server-side pool can determine the approximate size of the contents of the client-side pool, because at the moment the synchronization operation ends, the contents on the server side and client side are identical. The approximate size is important information for preparing data for the next synchronization on the server side.
(3) The server-side data pool can be used as a backup for the client-side data pool.

In a server, one data pool is prepared for each client. If the number of clients to be supported by one server is large, the server should have many data pools. However, this is not a big problem for the server by the following reasons:

(1) The server-side pool is the same size as the client-side pool. It is not particularly large for a server, because micro-clients have poor resources, while the server generally has rich resources.
(2) The server-side pools do not have to be placed in memory; instead, they can be placed on disk. They should be placed in memory only when they are accessed.

3.6 Synchronization Operation

Tuplink is based on Linda, but Linda assumes a permanent communication link. As explained in Section 2, we must not depend on a permanent communication link. The communication link may be intermittent, and almost all operations should be supported even while the link is disconnected.

We introduce the concept of synchronization in order to support operations while the link is disconnected. **Figure 2** shows an example of synchronization between a client and a server. At the time of (1), the contents of the two pools are identical. At the time of (2), the communication link is disconnected. Tuple D is created and tuple B is deleted on the client, while tuple E is created on the server. At the time of (3),

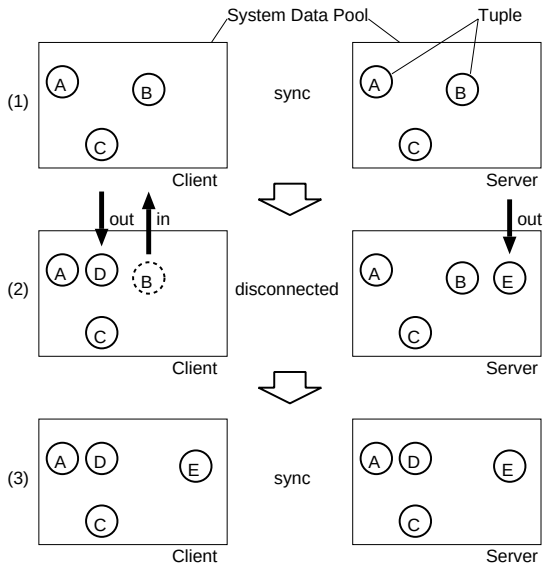

Fig. 2 Synchronization of client and server data pools.

the communication link is connected, and the two pools are synchronized.

The two connected system data pools are synchronized when the communication link is connected and their contents are made the same.

Because tuples are immutable, any change of the system data pool should consist of creation and/or deletion of tuples. The synchronization operation is performed in the following steps on both sides:

(1) Duplicate tuples that are created in the local data pool in the remote system data pool (remote tuple creation).
(2) Delete tuples that are deleted from the local data pool from the remote system data pool (remote tuple deletion).

This synchronization operation is performed continuously in connected mode.

3.7 Connection and Disconnection of a Communication Line

How a communication link is connected and disconnected is important in implementing a communication system. The optimal way of managing a communication link varies with the communication link cost, the volume of local storage, and the volume of data to be used. In Tuplink, a communication link is managed by sync, connect, and disconnect operations issued by applications. It should be emphasized that not only programs using the communication link, but any programs can call this API at any moment. All of the following methods should be supported:

(1) Manage the link according to the user's indications.
(2) Connect the link at a constant time interval.
(3) Manage the link according to the number and volume of tuples created or deleted in disconnected mode.
(4) Do not manage the link; let the other side, that is, the server, manages the link.

3.8 Semantics of Tuple Deletion

Because Tuplink supports "in" operations while the link is disconnected, the semantics of tuple deletion is changed from that of Linda. If two "in" operations for the same tuple at different nodes occur at the same time, one tuple can be deleted by two "in" operations. Therefore, a tuple cannot be used as a lock. This is a drawback of disconnect mode support, but it is not a serious problem in the case of Tuplink for the following reasons:

(1) Tuplink is not a language for distributed programming, like Linda.
(2) In Tuplink, two nodes—that is, the client and server—are asymmetric, and a specific tuple is usually deleted at only one node.

3.9 Buffer Sharing among Tuplink and Other System Components

The structure of a system using Tuplink is shown in Fig. 1. The data pool is used not only by the communication system of Tuplink, but also by other system components, such as file system, database system, or RPC system. All types of data of system components, such as HTML documents, RPC arguments, and people's addresses, are stored in tuples, and shared by all system components. Section 5 describes an example implementation of a database and file systems. Because other system components use these tuple as their buffers with no duplication, their implementations are expected to be compact.

4. Tuplink Protocol

We use a synchronization protocol, called the Tuplink protocol, in which each packet corresponds to a tuple.

4.1 Overview

In the Tuplink model, communication means synchronization of the two data pools. The function of a protocol for Tuplink is to synchronize the two data pools.

To allow efficient use of communication me-

dia, the Tuplink protocol is directly implemented on a data link layer, and provides reliable communication by using an unreliable data link layer.

Tuples are immutable, so the Tuplink protocol has two tasks: remote tuple creation and remote tuple deletion. Each side requests its party to create/delete tuples corresponding to locally created/deleted tuples. These tasks are necessary and sufficient for synchronizing two data pools. This is an important point for simplifying the protocol and implementation of Tuplink. The Tuplink protocol uses a tuple as a unit for maintaining reliability, and the reliability of the Tuplink protocol is implemented according to the following simple policies:

(1) The requester side of tuple creation and deletion repeatedly sends request packets until it receives the corresponding acknowledgment packet.
(2) The requested side sends one acknowledgment packet for each request packet.

We associate a state machine with each tuple. Each tuple keeps track of its own state changes. On the other hand, the underlying communication mechanism does not have an state for implementing reliable communication on lossy communication media.

4.2 Protocol Layers

The Tuplink system has the following three communication layers:

Tuplink protocol layer. Layer for managing remote tuple creation and deletion. This layer is the highest communication layer.

Packet ordering layer. Layer for ordering packets. This layer simply drops packets that are not sent in the correct order. Another role of this layer is to provide an interface that does not depend on the data link layer. We do not have to modify the Tuplink protocol layer, even if the underlying data link layer is changed. This layer also supports flow control, to limit the total size of data in the communication link without considering the order of data, if the underlying data link layer does not have a flow control mechanism. Since the sole aim of Tuplink's flow control mechanism is to support effective use of the data link layer, this mechanism is not necessary for a protocol with flow control, such as modem protocol V.34, but it is necessary for one with no flow control, such as the UDP/IP protocol.

Data link layer. Any data link layer that has a function for sending packets, such as UDP/IP. A reliable packet-sending function is not necessary in this layer.

4.3 Merits of the Tuple-based Protocol

We implemented a communication system that creates packets and sends them according to the state of a tuple. We call this approach using the state of the tuple itself the "Tuple-Based Protocol." It offers the following advantages:

(1) Memory for communication windows is not necessary, because the communication system can use tuples as communication windows.
(2) The effects of loss of packets are minimized, because the protocol does not guarantee the order in which all data are transferred, whereas stream-type communication does.
(3) In the case of sudden disconnection of the communication link, the action taken by the communication system is very simple: it simply stops transferring packets. When the communication link is reconnected, it simply restarts transferring packets. Because the synchronization state is held in each tuple, the communication system does not have to manage its own state.

4.4 State Machine of Connection

The state machine of connection is shown in **Fig. 3**. The initial state is WTCH (watching the link), in which the system waits for a connect request from a party. When the Tuplink system requests the data link layer to connect the link, the state of connection becomes a CALL (calling), and the data link layer attempts to connect the link. When party's connect request comes, the data link layer connects the link and changes the connection state to

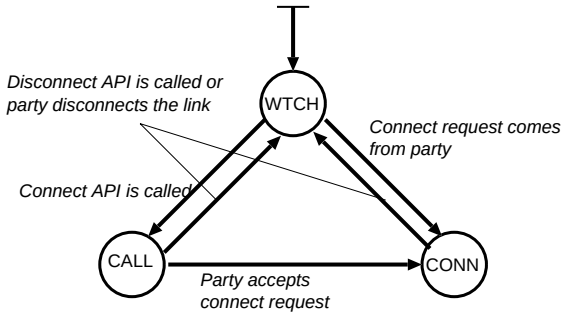

Fig. 3 State machine of connection.

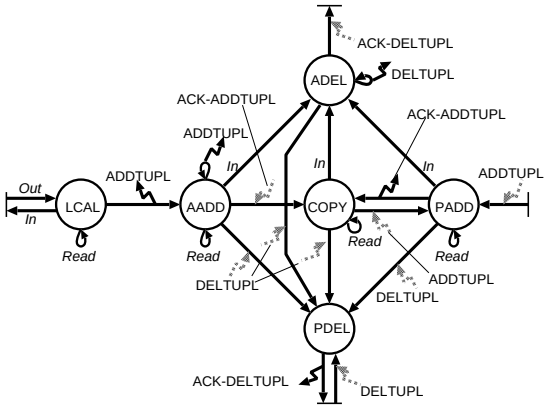

Fig. 4 State machine of each tuple.

CONN (connected).

4.5 State Machine of Each Tuple

The simple state machine of each tuple for implementing the above protocol is shown in **Fig. 4**.

In this figure, a circle denotes the state of a tuple, an italic string denotes a local operation of Tuplink, a straight arrow denotes a change of state, a folded solid arrow denotes the sending of a packet, and a folded dotted arrow denotes the receiving of a packet. For example, the LCAL state changes to the AADD state when an ADDTUPL packet is sent.

Next, we will describe the packets and states shown in Fig. 4.

4.6 Kinds of Packets

Tuplink protocol uses four kinds of packet, namely, ADDTUPL, ACK-ADDTUPL, DELTUPL, and ACK-DELTUPL, in order to execute remote tuple creation and remote tuple deletion. To create a tuple in the remote pool, the requester side repeatedly sends ADDTUPL packets until it receives a corresponding ACK-ADDTUPL packet. DELTUPL and ACK-DELTUPL are used similarly to delete a tuple in the remote pool. **Table 1** shows the kinds of packet.

Figure 5 shows the packet format of the Tuplink protocol. It has five fields, but the fifth field, "Tuple Contents," is used only when its packet type is ADDTUPL. The first field, "Packet Type," indicates the kind of packet. The second field, "Total Bytes," shows the total size in bytes of packets sent in this session. This field is used for both packet ordering and flow control if necessary. The third field is "Clean Flag." It becomes "true" if the remote pool has no packets to be sent. In case of the synchronization operation, the link is disconnected when the local pool has no packets to be sent and receive a packet whose "Clean Flag" is true. The fourth field, "TupleID," shows ID of Tuple. It is uniquely generated in both sides. The fifth field, "Tuple Contents," is used only for ADDTUPL packets, and packets of other types do not have this field. This field contains information for creating a tuple in the remote pool.

Table 1 Kinds of packets.

ADDTUPL	Packet for creating a tuple in a remote data pool. It includes the ID of the tuple to be created and necessary information on all items of the tuple.
ACK-ADDTUPL	Packet for acknowledging an ADDTUPL Packet. It includes the ID of the created tuple.
DELTUPL	Packet for deleting a tuple from a remote data pool. It includes the ID of the tuple to be deleted.
ACK-DELTUPL	Packet for an acknowledging a DELTUPL Packet. It includes the ID of the deleted tuple.

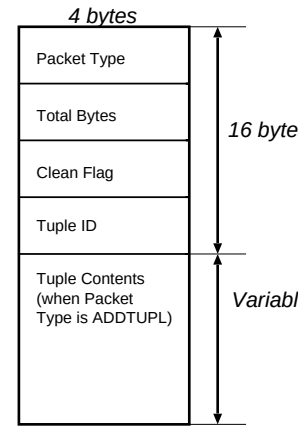

Fig. 5 Tuplink packet.

4.7 States and Actions

Each tuple has six possible states to manage the sending and receiving of its corresponding packets. A tuple created in the local pool has LCAL state. When the communication system sends an ADDTUPL packet for the tuple, its state becomes AADD (active adding). If the communication system receives an ACK-ADDTUPL packet successfully, the state of the tuple changes from AADD to COPY (replicated). Similar state changes occur in the case of a tuple deletion. The six possible states of a tuple and the corresponding actions are shown in **Table 2**.

Table 2 States and actions.

LCAL	The initial state of a tuple created by an "out" operation in a local host. It changes to AADD when an ADDTUPL packet is sent. Tuples with LCAL state exist only in a local data pool, and there are no corresponding tuples in a remote data pool, so the communication system can simply delete them without sending packets.
AADD	The second state of a tuple created by an "out" operation in a local host. One or more ADDTUPL packets of the tuple have been sent, but no corresponding ACK-ADDTUPL packet has been received yet. This state changes to COPY when a corresponding ACK-ADDTUPL packet is received.
PADD	The state of a tuple created by an ADDTUPL packet. Tuples with this state are created by receiving ADDTUPL packets. This state changes to COPY when an ACK-ADDTUPL packet is sent to acknowledge receipt of an ADDTUPL packet.
COPY	The state of a tuple which exists in both data pools. This is the synchronized state, but it may changes to PADD so that an ACK-DELTUPL packet can be sent again. The synchronization procedure is finished when all tuples have this state.
ADEL	The state of a tuple deleted by an "in" operation in a local host. One or more DELTUPL packets for a tuple with this state will be sent to a remote host. A tuple with this state is erased when a corresponding ACK-DELTUPL packet is received.
PDEL	The state of a tuple that has received a corresponding DELTUPL packet. A tuple with this state is erased when an ACK-DELTUPL packet is sent. When a DELTUPL packet for the erased tuple arrives, a new tuple with the PDEL state is created temporarily in order to send an ACK-DELTUPL for the received packet.

The Tuplink protocol delays as long as possible the transfer of packets whose contents are the same as those of previously sent packets. The retransmission occurs only after all other remaining work is finished. In the case of micro-clients, this delay is effective for eliminating unnecessary transfer of packets, because unreliable communication media are often used.

5. Prototype Implementation

We implemented prototype systems using Tuplink on several platforms. The implementations of these prototype systems are described in this section.

5.1 Platforms

The prototype systems are implemented on the following platforms.

Windows NT/Windows 95
The server parts of the prototype systems are implemented on this platform.

Windows CE and PalmPilot/Palm III
A compact implementation of Lotus Notes using Tuplink, called "micro-Notes," is implemented. Compiled forms of the definitions of both Screens and scripts written in LotusScript are downloaded and interpreted by the microNotes. A compact implementation of Lotus Notes using Tuplink runs on the system.

DataScope (Smart Phone)
DataScope is a phone-shaped PDA [6], and it has a 16-bit processor, 128 KB of RAM, a phone, a modem, and a quarter-VGA display. **Figure 6** is a picture of DataScope. Its price is about $200 in Japan. PIM applications for accessing Lotus Notes by using Tuplink are implemented.

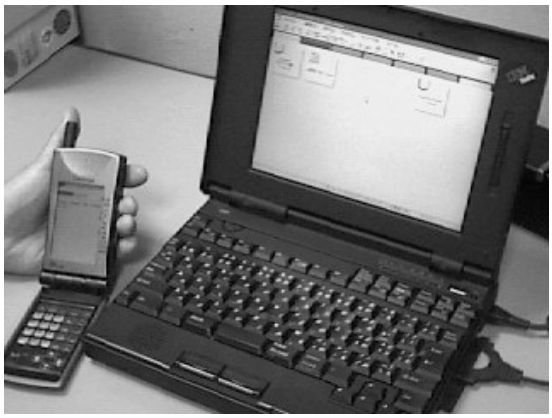

Fig. 6 DataScope (left) and Notebook PC (right).

These prototype systems all have similar functions; in this section we will mainly describe the prototype system on DataScope.

5.2 Basic Structure

Figure 7 shows modules of the prototype system on DataScope.

The shaded parts are newly implemented, and the others are the original programs.

Tuplink is implemented on both server and client sides. The Notes-Tuplink bridge on the server side is a module for connecting databases of Lotus Notes and Tuplink. The Database Emulator on the client side provides an API for the existing database system, so we do not have to modify applications that use the API, such as

the scheduler and address book. We also implemented an application for handling Lotus Notes mail on the client side.

5.3 API of the Tuplink System

The Tuplink system API is defined in the C language. The reason we do not use the C++ language is that compilers for micro-devices do not support the C++ language in many cases, and C++ binaries tend to become large. This API is categorized into three classes, namely, tuple objects API, matching conditions API, and data pool API. **Table 3** shows the Tuplink system API. The API uses three types of objects. Tuple objects are introduced to handle tuples to be created or retrieved tuples. Condition objects are introduced to handle matching conditions. Value objects are introduced to handle each item's value and type. In this table, the "Tupl" structure is for tuple objects, the "Cond" structure is for condition objects, and the "value" structure is for value objects.

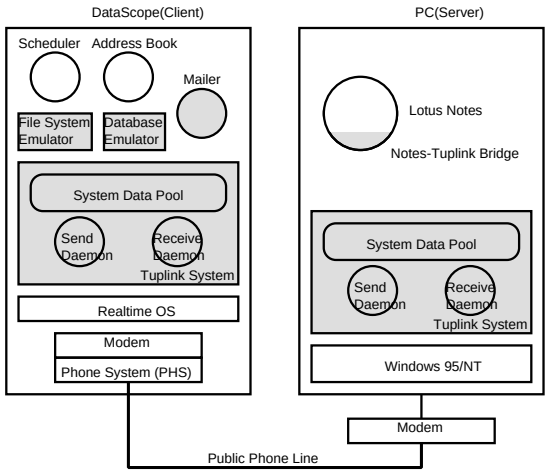

Fig. 7 Modules on DataScope.

To create a tuple, an application first uses a tuple objects API, such as TuplNew or TuplAddItem, in order to create a tuple object, and then uses the TuplinkOut API to create a tuple in the data pool. To retrieve a tuple, an application first uses a condition objects API, such as CondNew or CondAddItem, to create a condition object, and then uses the TuplinkIn API with the condition object to retrieve a tuple in the data pool as a tuple object. It then uses a tuple objects API to access the retrieved tuple object. It uses the TuplGetTags API for getting all tag strings of the tuple, and the TuplGetVal API to get each item's value and type with the specified tag strings.

5.4 Implementation of the Tuplink System

Tuplink consists of a library for the Tuplink

Table 3 Tuplink system API.

Tuple Objects API	
`Tupl *TuplNew()`	Create tuple object
`int TuplDel(Tupl *)`	Free tuple object
`int TuplAddItem(Tupl *, char *, value *)`	Add item to tuple object
`int TuplDelItem(Tupl *, char *)`	Delete item from tuple object
`int TuplGetVal(Tupl *, char *, value *)`	Get value of item with a specified tag
`int TuplGetTags(Tupl *, char **, int)`	Get array of tag strings
Matching Condition Objects API	
`Cond *CondNew()`	Create matching condition object
`int CondDel(Cond *)`	Free matching condition object
`int CondAddItem(Cond *, char *, value *)`	Add item to object
`int CondDelItem(Cond *, char *)`	Delete item to object
Data Pool API	
`int TuplinkInit()`	Initialize Tuplink system
`int TuplinkTerm()`	Terminate Tuplink system
`int TuplinkSync(char *)`	Synchronize data pools
`int TuplinkConnect(char *)`	Start synchronization
`int TuplinkDisconnect()`	Stop synchronization
`int TuplinkOut(Tupl *)`	Create tuple in data pool
`Tupl *TuplinkRead(Cond **, int)`	Refer to tuple in data pool (blocking)
`Tupl *TuplinkNbRead(Cond **, int)`	Refer to tuple in data pool (non-blocking)
`int TuplinkMultiRead(Cond **, int, Tupl **, int)`	Refer to multiple tuples in data pool (non-blocking)
`Tupl *TuplinkIn(Cond **, int)`	Retrieve tuple in data pool (blocking)
`Tupl *TuplinkNbIn(Cond **, int)`	Retrieve tuple in data pool (non-blocking)
`int TuplinkMultiIn(Cond **, int, Tupl **, int)`	Retrieve multiple tuples in data pool (non-blocking)

API and two threads. One thread is for sending packets, and the other is for receiving packets.

A system data pool consists of three hash tables: one for ID-based search, another for tag-string-based search, and a third for string-value-based search. All tag strings with the same value in a data pool are shared, and all string values with the same value in a data pool are also shared.

5.5 Database System

The database system is implemented by using Tuplink. A write conflict occurs when the same data are modified by both server- and client-side programs at the same time. To avoid this problem, a simple optimistic replication control mechanism is implemented. The following example shows a tuple representing one person's address information:

```
["Service"="Address"(STR),
 "Created"="Server"(STR),
 "Form"="Person"(STR),
 "FirstName"="Michael"(STR),
 "LastName"="Jackson"(STR),
 "PhoneNumber"="123-4567"(STR)]
```

The item "Created"="Server"(STR) means that this tuple is created by the server side, and the item "Created"="Client" means that this tuple is created by the client side. In this database system, a tuple is deleted only on the server side. If data are modified, the client side creates a new tuple for the modified data, and does not delete the original tuple. If data are referred to, the client side first searches for a tuple created by the client side, and then searches for a tuple created by the server. The server side always searches for a tuple created by the client. If it finds one, it creates a new tuple with the modified data, and deletes both the original and modified tuples.

5.6 File System

The file system is implemented by using Tuplink. In this system, one file is mapped to one tuple. The following is an example:

```
["FileName" = "/tmp/sample"(STR),
 "FileSize" = 26(NUM),
 "Version"  = 1(NUM),
 "Contents" = "sample contents."(STR)]
```

A simple optimistic replication control mechanism similar to a database system is implemented.

6. Discussion

Using the prototype implementation, we have measured several aspects of the Tuplink system from the viewpoint of the three requirements described in Section 2.

6.1 Limited Resource Support

In terms of usage, memory can be categorized into two types: memory needed for the program code, and memory needed for the data to be used by the code.

We measured the amounts of memory needed for the program code. In the Tuplink system implemented on the DataScope, this is about 32 KB, including all necessary programs, such as a protocol handler and a device driver for the modem. The code size is small enough to meet our target total memory size of several hundred kilobytes for Tuplink.

Next, we estimate the amounts of memory needed for data to be used by the code. We cannot yet directly compare the amounts of memory needed for Tuplink data and for the data of the existing system. However, we can show that network features can be easily added to some existing applications that access only local databases.

The existing address book application, which is preloaded in DataScope, accesses only local databases. It can handle up to 100 persons' addresses. We changed the local database into a database that uses Tuplink. By this change, the application can access remote database of Lotus Notes without modification.

Figure 8 shows a graph of the memory use of the application in both original and Tuplink cases.

In Fig. 8, the X-axis represents the number of people for whom data are input. The Y-axis represents the amount of memory used. The dotted line in the figure shows the amount of memory used in the original address book application of DataScope. The solid line shows the amount of memory used in Tuplink that manages the same information as the original application. To calculate the volume, we as-

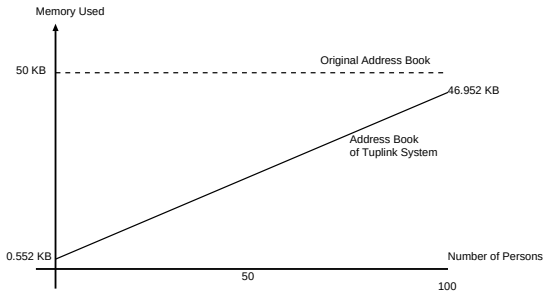

Fig. 8 Memory use of an address book application.

sume that half of the fields for each person are filled.

Figure 8 shows that memory use of the new extended system is on almost the same level as that of the existing application. In addition, it must be stressed that Tuplink does not need extra memory for communication buffers.

6.2 Disconnected Mode Support

The Tuplink system also provides good support for disconnected operations. Various kinds of applications, such as an address book, a mail manager, and a document-editing system, are made operable in disconnected mode. For example, in the case of the address book application of DataScope that accesses local databases, we replaced the existing local database with a database using Tuplink. As a result, the existing application can access remote databases of Lotus Notes without any modifications, and users can run the application while the link is disconnected. Note that the existing application itself is not modified at all.

6.3 Low-Quality Link Support

Next, we evaluate the efficiency of the Tuplink protocol. To simplify the discussion, we compare the throughput of the Tuplink protocol and the TCP/IP protocol, although the Tuplink protocol is a synchronization protocol, not a stream-based protocol like the TCP/IP protocol.

We measured the time taken to transfer 18 KB of data between two PCs connected by two 33.6-kbps modems and a telephone line emulator. All compression was disabled, to compare the bare performance of the two protocols. The system using the Tuplink protocol transferred one hundred 180-byte tuples, while the system using the TCP/IP protocol simply transferred 18 KB of data.

The Windows 95 operating system was used to measure the performance of the Tuplink protocol, and the Unix operating system (FreeBSD 2.2.2R) was used to measure the performance of the TCP/IP protocol. We modified each system's program code for receiving packets in order to control the error rate of packet transfer.

Figure 9 shows the transfer times using the Tuplink protocol and the TCP/IP protocol. In this figure, the X-axis represents the error rate of packet transfer, and the Y-axis represents the transfer time in seconds.

The solid line shows the time for the Tuplink protocol, and the dotted line shows the time for the TCP/IP protocol. The results for both

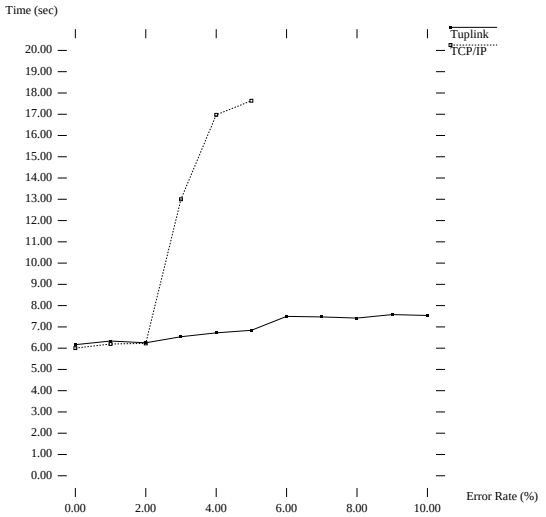

Fig. 9 Transfer time.

protocols are almost the same at an error rate of from 0 to 2%, and their throughputs are about 24 kbps. This value, about 24 kbps, is good enough because all protocols have necessary overhead for implementing their functions, but the line of the TCP/IP protocol rises rapidly at an error rate of 3%. In contrast, the line of the Tuplink protocol rises gradually. This shows the robustness of the Tuplink protocol on low-quality links.

The reason that TCP/IP protocol's performance decreases dramatically at an error rate of 3% is TCP/IP's congestion control mechanism. The congestion control is started by even a single packet loss, because almost all packet losses are caused by buffer overflow on gateway machines when LANs are used. However, it causes too much performance loss if the error rate becomes high.

In addition, Tuplink can work with sudden disconnections of communication links. Applications can recover from the sudden disconnections by simply restarting the synchronization operation when the link is reconnected, because the communication system itself does not have any state.

7. Related Technologies

7.1 System Support for Compact Mobile Clients

Xerox's PARCTAB system [7] was one of the earliest studies of the implementation and application of compact clients connected to a network. The implementations of all services are

off-loaded to the server by using the system's "Tab Agent" mechanism. A device is assumed to be permanently connected to a network, and a client does not have a local context.

InfoPad [8], which was implemented by a group at U.C. Berkeley, is another example of the study of compact network clients. It also assumes permanent connection, and processes are off-loaded by a "Type Server."

These studies assume permanent connection, whereas, our study focuses on intermittent communication links.

There have been many studies of various system service designs for mobile computing.

A typical example is support of a local context with a weakly connected client. The semantics and implementation of weak consistency have been studied by a number of researchers [9]. Bayou [10] applied a technology of this type to mobile computing. It implements a mechanism for readjusting conflict that requires intervention from applications on multiple nodes, and for enabling client applications to continue even if a conflict remains.

Rover [11] implemented a check-in-and-check-out style consistency control mechanism for actual mobile applications by using relocatable dynamic objects.

Support for mobile computing is also provided by a robust RPC mechanism. The message queue (MQ) product implements a transactional queuing mechanism over a network [12]. Rover implemented a queued remote procedure call (QRPC), which supports persistent queuing of RPC requests to cope with disconnected mode.

Like our study, these studies do not assume permanent connection, and propose new functions for supporting some kinds of mobility. The aim of our study is to explore a system structure for eliminating data copying, and not to propose a new function. Our study's goal is to support every function for supporting mobility including the existing functions, such as RPC or distributed database, by our system.

Cache management is another topic. Much work has been done on supporting WWW browsing in a mobile environment. WebExpress [13] implements an HTML cache and provides an original transfer mechanism to reduce network traffic and improve response.

Tuplink depends on all these preceding studies. It is especially closely related to:

- Efficient implementations of protocol stacks [14),15)]
- Protocols for communications between weakly connected nodes
- Replication mechanisms for weakly connected storage systems

Our approach provides a framework for implementing system support for distributed processing by weakly connected nodes in a minimal set of memory and processor resources.

7.2 Studies of Linda and Its Applications

Linda is still being studied, and has been used in various kinds of applications by many researchers since it was proposed in 1985. Carriero and Gelernter [16] used the mechanism to implement applications for a parallel processor. A kernel supporting Linda's communication was implemented for this purpose.

Ahuja, et al. [17] implemented a dedicated machine to execute Linda.

Schoenfeldinger [18] used Linda to implement a CGI program in a WWW server. It simplifies the handling of the state of requests, which is not adequately supported by the original CGI mechanism.

The concept of data pools has been used to implement autonomous distributed systems. The Information Bus [19] implemented a shared data pool and a mechanism for publishing and scribing data to and from the pool. It is used to implement a highly reliable system that can be operated 24 hours a day, 7 days a week.

JavaSpaces [20] uses a concept similar to Linda. It is strongly coupled with the Java language, and is used for distributed computing in heterogeneous environments.

The most basic concept of Linda is Tuple Space, and all of these studies assume permanent connection to support a globally shared tuple space. Our model is based on the Tuple Space concept, but the model differs from the original one. In our model, the contents of space at nodes are not always identical. Our model introduces explicit synchronization control mechanism in order to support a weakly connected client/server system with minimal resources.

8. Conclusion

Micro-clients explore new possibilities in network computing. The key is a technology that drastically reduces the cost of the system, thus allowing the use of a much larger number of applications than in network computing on PCs.

We proposed an approach for building suit-

able system software for micro-clients. A single data pool, called a system data pool, is used to keep all system and user data, thus eliminating data copying by system components, reducing the working memory size, and reducing functional duplication between system components. Copies of the system data pools are provided on both the client and server sides, and communication between client and server nodes is an operation for synchronizing these copies. The abstraction helps to simplify application programs, and helps to implement reliable communication in an efficient way.

We built the Tuplink model on the basis of the Linda Tuple Space concept to implement system data pools. Our experience in the system implementation shows that Tuplink can effectively provide various kinds of system services.

References

1) Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R. and Yelick, K.: Intelligent RAM (IRAM): Chips that Remember and Compute, *Proc. International Solid-State Circuits Conference*, IEEE (1997).
2) Papadopoulos, C. and Parulkar, G.M.: Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation, *IEEE/ACM Trans. Networking*, Vol.1, No.2, pp.199–216 (1993).
3) Kay, J. and Pasquale, J.: The Importance of Non-data-touching Processing Overheads in TCP/IP, *Proc. SIGCOMM '93*, ACM (1993).
4) Gelernter, D.: Generative Communication in Linda, *ACM Trans. Programming Languages and Systems*, Vol.7, No.1, pp.80–112 (1985).
5) Goodman, D.: *JavaScript Bible*, Second Edition, IDG Books (1996).
6) Kyocera, Inc: DataScope, http://www.kyocera.co.jp/frame/product/telecom/english/d_scope/index.htm.
7) Want, R., Schilit, B.N., Adams, N.I., Gold, R., Petersen, K., Goldberg, D., Ellis, J.R. and Weiser, M.: An Overview of the PARCTAB Ubiquitous Computing Experiment, *Personal Communications*, Vol.2, No.6, pp.28–43 (1995).
8) Narayanaswamy, S., Seshan, S., Amir, E., Brewer, E., Brodersen, R.W., Burghardt, F., Burstein, A., Chang, Y.-C., Fox, A., Gilbert, J.M., Han, R., Katz, R.H., Long, A.C., Messerschmitt, D.G. and Rabaey, J.M.: Application and Network Support for InfoPad, *Personal Communications*, Vol.3, No.2, pp.4–17 (1996).
9) Davaidson, S.B., Garcia-Molina, H. and Skeen, D.: Consistency in Partitioned Networks, *Comput. Surv.*, Vol.17, No.3, pp.341–370 (1985).
10) Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J. and Hauser, C.H.: Managing Update Conflicts in Bayou, A Weakly Connected Replicated Storage System, *Proc. 15th ACM Symposium on Operating System Principles*, ACM (1995).
11) Joseph, A.D., deLespinasse, A.F., Tauber, J.A., Gifford, D.K. and Frans Kaashoek, M.: Rover: A Toolkit for Mobile Information Access, *Proc. 15th ACM Symposium on Operating System Principles*, ACM (1995).
12) IBM Manual, MQSeries Distributed Queuing Guide, SC33-1139-06 (1995).
13) Housel, B.C. and Lindquist, D.B.: WebExpress: A System for Optimizing Web Browsing in a Wireless Environment, *Proc. Second Annual International Conference on Mobile Computing and Networking*, ACM/IEEE (1996).
14) Clark, D., Jacobson, V., Romkey, J. and Salwen, H.: An Analysis of TCP Processing Overhead, *Communications Magazine*, Vol.27, No.6, pp.23–29 (1989).
15) Key, J. and Pasquale, J.: Measurement, Analysis, and Improvement of UDP/IP Throughput for DECstation 5000, *Proc. Winter USENIX*, USENIX (1993).
16) Carriero, N. and Gelernter, D.: The S/Net's Linda Kernel, *ACM Trans. Comput. Syst.*, Vol.4, No.2, pp.110–129 (1986).
17) Ahuja, S., Carriero, N.J., Gelernter, D.H. and Krishnaswamy, V.: Matching Language and Hardware for Parallel Computation in the Linda Machine, *IEEE Trans. Comput.*, Vol.37, No.8, pp.921–929 (1988).
18) Schoenfeldinger, W.J.: WWW Meets Linda: Linda for Global WWW-Based Transaction Processing Systems, *World Wide Web Journal*, Issue 1, http://www.w3j.com/*1*/schoen.174/paper/174.html (1995).
19) Oki, O., Pfluegl, M., Siegel, A. and Skeen, D.: The Information Bus(R): An Architecture for Extensible Distributed Systems, *Proc. 14th ACM Symposium on Operating System Principles*, ACM (1993).
20) SUN Microsystems, Inc.: JavaSpaces Specification, http://www.javasoft.com/products/jini/specs (1998).

(Received February 5, 1999)
(Accepted July 5, 2000)

Yasushi Negishi received his B.E. and M.E. degrees from Tokyo Institute of Technology in 1987 and 1989, respectively. Since April 1989, he is a researcher of IBM Tokyo Research Laboratory. His current interests include performance of communicaiton systems and protocols. He is a member of ACM and IEEE Computer Society.

Kiyokuni Kawachiya received his B.S. and M.S. degrees from the University of Tokyo in 1985 and 1987, respectively. Since April 1987, he is a researcher of IBM Tokyo Research Laboratory and have been working on multiprocessor operating systems, multimedia processing systems, mobile information systems, and Java just-in-time compiler. He received IPSJ convention award in 1994. His main interest is system software area including operating systems and programming language.

Hiroki Murata received his B.E. and M.E. degrees from Waseda University, Tokyo, Japan, in 1987 and 1989, respectively. Since April 1989, he is a researcher of IBM Tokyo Research Laboratory. His current research interests include operating systems, communications, and user interface.

Kazuya Tago received his Dr.E. degree from University of Tsukuba in 1986. He joined Tokyo Research Laboratory of IBM Japan in 1990 after working at University of Tsukuba and University of Tokyo as a research associate. His current research interests include design approaches of operating systems and communication systems. He is a member of Information Processing Society of Japan. He received the best paper award of IPSJ in 1984.